

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



ios Apprentice

FIFTH EDITION

Tutorial 4: StoreSearch

By Matthijs Hollemans

iOS Apprentice

Matthijs Hollemans

Copyright ©2016 Razeware LLC.

Notice of Rights

All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written permission of the copyright owner.

Notice of Liability

This book and all corresponding materials (such as source code) are provided on an “as is” basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in action of contract, tort or otherwise, arising from, out of or in connection with the software or the use of other dealing in the software.

Trademarks

All trademarks and registered trademarks appearing in this book are the property of their own respective owners.

License

By purchasing *iOS Apprentice*, you have the following license:

- You are allowed to use and/or modify the source code in *iOS Apprentice* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *iOS Apprentice* in as many apps as you want, but must include this attribution line somewhere inside your app: “Artwork/images/designs: from *iOS Apprentice* book, available at www.raywenderlich.com”.
- The source code included in *iOS Apprentice* is for your personal use only. You are NOT allowed to distribute or sell the source code in *iOS Apprentice* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

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: Extended

| | |
|-------------------------------------|-----|
| Tutorial 4: StoreSearch | 6 |
| The StoreSearch app | 6 |
| In the beginning... .. | 8 |
| Custom table cells and nibs | 31 |
| The debugger..... | 50 |
| It's all about the networking | 58 |
| Asynchronous networking | 87 |
| URLSession..... | 99 |
| The Detail pop-up..... | 128 |
| Fun with landscape | 168 |
| Refactoring the search | 200 |
| Internationalization | 222 |
| The iPad | 243 |
| Distributing the app | 273 |
| The end..... | 289 |

Tutorial 4: StoreSearch

By Matthijs Hollemans

The StoreSearch app

One of the most common things that mobile apps do is talking to a server on the internet. It's beyond question: if you're writing mobile apps, you need to know how to upload and download data.

In this lesson you'll learn how to do HTTP GET requests to a web service, how to parse JSON data, and how to download files such as images.

You are going to build an app that lets you search the iTunes store. Of course, your iPhone already has apps for that ("App Store" and "iTunes Store" to name two), but what's the harm in writing another one?

Apple has made a web service available for searching the entire iTunes store and you'll be using that to learn about networking.

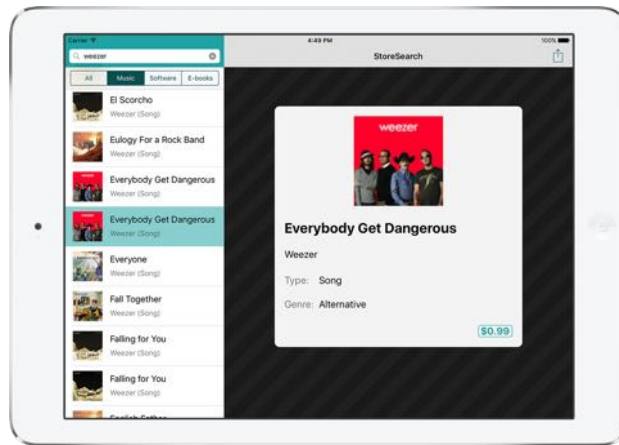
The finished app will look like this:



The finished StoreSearch app

You will add search capability to your old friend, the table view. There is an animated pop-up with extra information when you tap an item in the table. And when you flip the iPhone over to landscape, the layout of the app completely changes to show the search results in a different way.

There is also an iPad version of the app:



The app on the iPad

The to-do list for building **StoreSearch** is roughly as follows:

- Create a table view (yes, again!) with a search bar.
- Perform a search on the iTunes store using their web service.
- Understand the response from the iTunes store and put the search results into the table view.
- Each search result has an artwork image associated with it. You'll need to download these images separately and place them in the table view as well.
- Add the pop-up screen with extra info that appears when you touch an item.
- When you flip to landscape, the whole user interface changes and you'll show all of the icons in a paging scroll view.
- Add support for other languages. Having your app available in languages besides English dramatically increases its audience.
- Make the app universal so it runs on the iPad.

This tutorial fills in the missing pieces and rounds off the knowledge you have obtained from the previous tutorials.

You will also learn how to distribute your app to beta testers with so-called Ad Hoc Distribution, and how to submit it to the App Store.

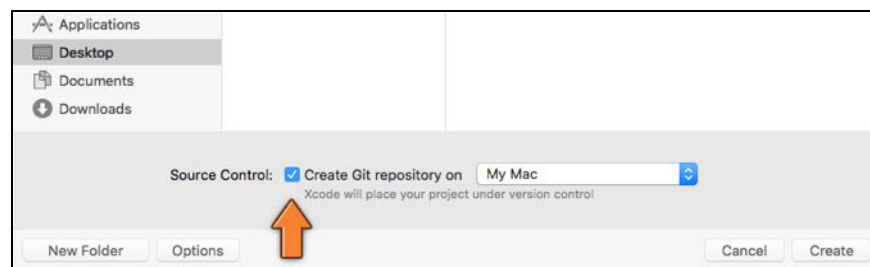
There's a lot of work ahead, so let's get started!

In the beginning...

Fire up Xcode and make a new project. Choose the **Single View Application** template and fill in the options as follows:

- Product Name: **StoreSearch**
- Organization Name: your name
- Organization Identifier: com.yourname
- Language: **Swift**
- Devices: **iPhone**
- Use Core Data, Include Unit Tests, Include UI Tests: leave these unchecked

When you save the project Xcode gives you the option to create a so-called **Git repository**. You've ignored this option thus far but now you should enable it:



Creating a Git repository for the project

If you don't see this option, click the Options button in the bottom-left corner.



Git and version control

Git is a so-called **revision control system**. In short, Git allows you to make snapshots of your work so you can always go back later and see a history of what you did. Even better, a tool such as Git allows you to work on the same code with multiple people.

Imagine what happens if two programmers changed the same source file at the same time. Things will go horribly wrong! It's quite likely your changes will accidentally be overwritten by a colleague's. I once had a job where I had to shout

down the hall to another programmer, “Are you using file X?” just so we wouldn’t be destroying each other’s work.

With a revision control system such as Git, each programmer can work independently on the same files, without a fear of undoing the work of others. Git is smart enough to automatically merge all of the changes, and if there are any conflicting edits it will let you resolve them before breaking anything.

Git is not the only revision control system out there but it’s the most popular one for iOS. A lot of iOS developers share their source code on GitHub (github.com), a free collaboration site that uses Git as its engine. Another popular system is Subversion, often abbreviated as SVN. Xcode has built-in support for both Git and Subversion.

In this tutorial I’ll show you some of the basics of using Git. Even if you work alone and don’t have to worry about other programmers messing up your code, it still makes sense to use it. After all, you might be the one messing up your own code and with Git you’ll always have a way to go back to your old – working! – version of the code.



The first screen in StoreSearch will have a table view with a search bar, so let’s make the view controller for that screen.

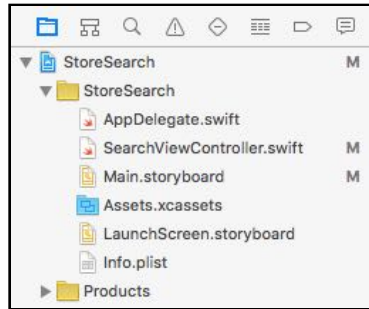
- In the project navigator, rename **ViewController** to **SearchViewController**.
- In **SearchViewController.swift**, change the class line to:

```
class SearchViewController: UIViewController {
```

Note that this is a plain `UIViewController`, not a table view controller.

- In the storyboard, change the **Custom Class** for the view controller to **SearchViewController** (in the Identity inspector).
- For good measure, run the app to make sure everything works. You should see a white screen with the status bar on top.

Notice that the project navigator now shows **M** (and possibly **A**) icons next to some of the filenames in the list:



Xcode shows the files that are modified

If you don't see these icons, then choose the **Source Control → Refresh Status** option from the Xcode menu bar. (If that gives an error message or still doesn't work, simply restart Xcode. That's a good tip in general: if Xcode is acting weird, restart it.)

An M means the file has been modified since the last "commit" and an A means this is a file that has been added since then.

So what is a commit?

When you use a revision control system such as Git, you're supposed to make a snapshot every so often. Usually you'll do that after you've added a new feature to your app or when you've fixed a bug, or whenever you feel like you've made changes that you want to keep. That is called a **commit**.

When you created the project, Xcode made the initial commit. You can see that in the Project History window.

➤ Choose **Source Control → History...**

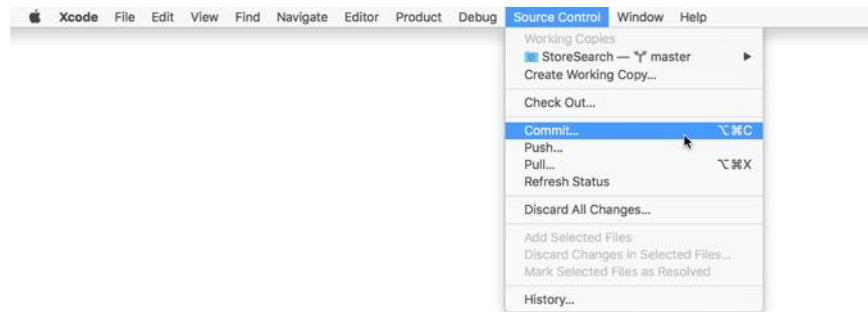


The history of commits for this project

You may get a popup at this point asking for permission to access your contacts. That allows Xcode to add contact information to the names in the commit history. This can be useful if you're collaborating with other developers. You can always change this later under Security & Privacy in System Preferences.

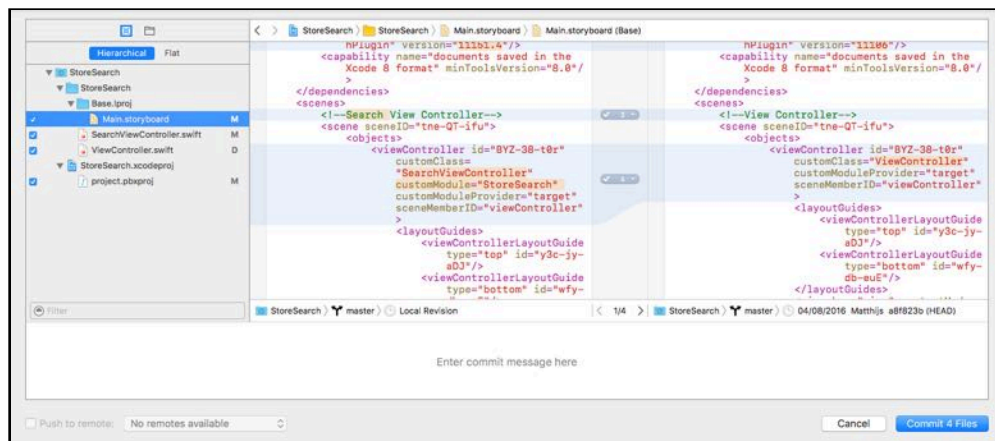
➤ Let's commit the change you just made. Close the history window. From the

Source Control menu, choose **Commit**:



The Commit menu option

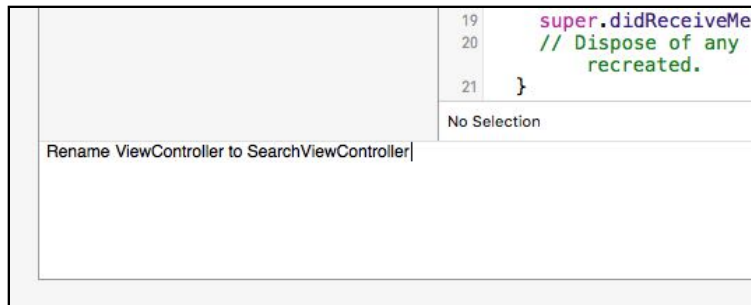
This opens a new window that shows in detail what changes you made. This a good time to quickly review the differences, just to make sure you're not committing anything you didn't intend to:



Xcode shows the changes you've made since the last commit

It's always a good idea to write a short but clear reason for the commit in the text box at the bottom. Having a good description here will help you later to find specific commits in your project's history.

► Write: **Rename ViewController to SearchViewController**

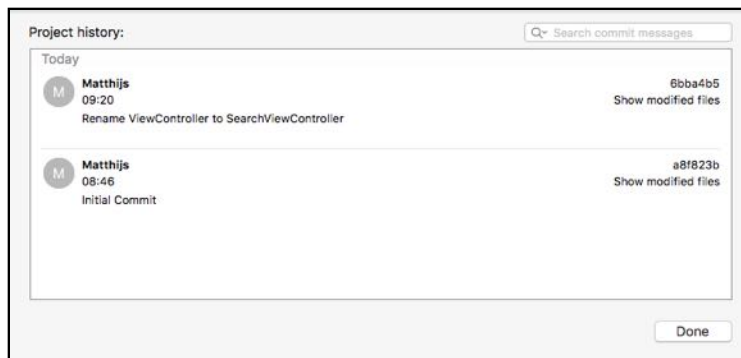


Writing the commit message

► Press the **Commit 4 Files** button. You'll see that in the project navigator the M and A icons are gone (at least until you make the next change).

If you're wondering why it said "Commit 4 Files", renaming ViewController.swift counts as two – deleting the old file and adding it back with the new name – so in total four files were modified.

The **Source Control** → **History** window now shows two commits:

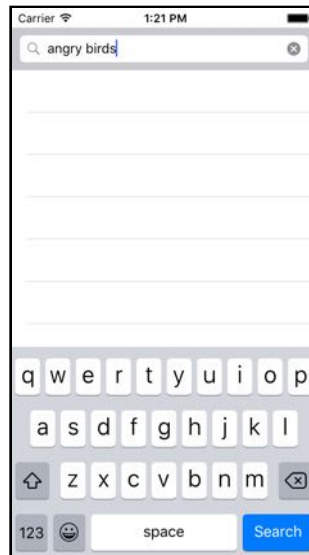


Your commit is listed in the project history

If you click **Show modified files**, Xcode will show you what has changed with that commit. You'll be doing commits on a regular basis and by the end of the tutorial you'll be a pro at it.

Creating the UI

The app still doesn't do much yet. In this section, you'll build the UI to look like this, a search bar on top of a table view:



The app with a search bar and table view

Even though this screen uses the familiar table view, it is not a *table* view controller but a regular `UIViewController`.

You are not required to use a `UITableViewController` if you have a table view. For this app I will show you how to do without.



UITableViewController vs. UIViewController

So what exactly is the difference between a *table* view controller and a regular view controller?

First off, `UITableViewController` is a subclass of `UIViewController` so it can do everything that a regular view controller can. However, it is optimized for use with table views and has some cool extra features.

For example, when a table cell contains a text field, tapping that text field will bring up the on-screen keyboard. `UITableViewController` automatically scrolls the cells out of the way of the keyboard so you can always see what you're typing.

You don't get that behavior for free with a plain `UIViewController`, so if you want this feature you'll have to program it yourself.

`UITableViewController` does have a big restriction: its main view must be a `UITableView` that takes up the entire screen space (except for a possible navigation bar at the top, and a toolbar or tab bar at the bottom).

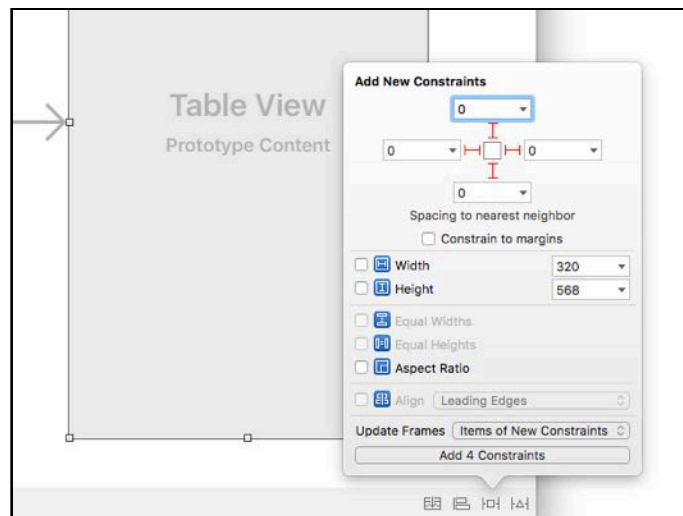
If your screen consists of just a `UITableView`, then it makes sense to make it a

UITableViewController. But if you want to have other views as well, the more basic UIViewController is your only option.

That's the reason you're not using a UITableViewController in this app. Beside the table view the app has another view, a UISearchBar. It is possible to put the search bar *inside* the table view as a special header view, but for this app it will always be sitting on top.



- Open the storyboard and use the **View as:** panel to switch to the **iPhone SE** dimensions. It doesn't really matter which iPhone model you choose here, but the iPhone SE makes it easiest to follow along with this book.
- Drag a new **Table View** into the view controller.
- Make the Table View as big as the main view (320 by 568 points) and then use the **Pin menu** at the bottom to attach the Table View to the edges of the screen:



Creating constraints to pin the Table View

Remember how this works? This app uses Auto Layout, which you learned about in the Bull's Eye and Checklists tutorials. With Auto Layout you create **constraints** that determine how big the views are and where they go on the screen.

- First, uncheck **Constrain to margins**. Each screen has 16-point margins on the left and right (although you can change their size). When "Constrain to margins" is enabled you're pinning to these margins. That's no good here; you want to pin the Table View to the edge of the screen instead.
- In the **Spacing to nearest neighbor** section, select the red bars to make four

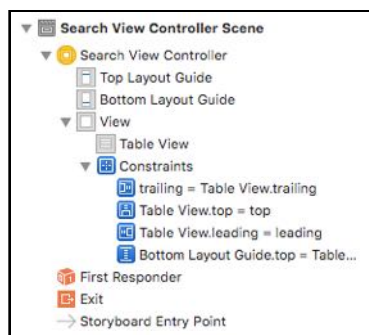
constraints, one on each side of the Table View. Keep the spacing values at 0.

This pins the Table View to the edges of its superview. Now the table will always fill up the entire screen, regardless of whether you're running the app on a 3.5-inch or a 5.5-inch device.

► Make sure **Update Frames** says **Items of New Constraints**. This moves the view to the position dictated by the new constraints, if necessary.

► Click the **Add 4 Constraints** button to finish.

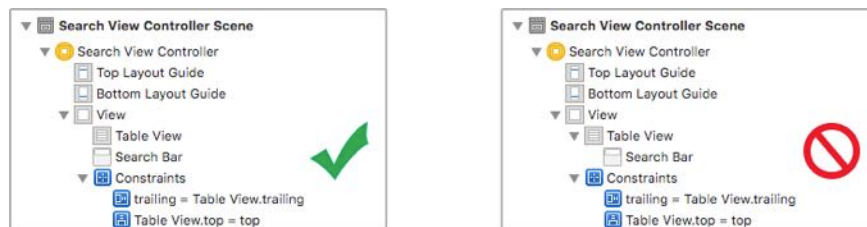
If you were successful, there are now four blue bars surrounding the table view, one for each constraint. In the outline pane there is also a new Constraints section:



The new constraints in the outline pane

► From the Object Library, drag a **Search Bar** into the view. (Be careful to pick the Search Bar and not "Search Bar and Search Display Controller".) Place it at Y = 20 so it sits right under the status bar.

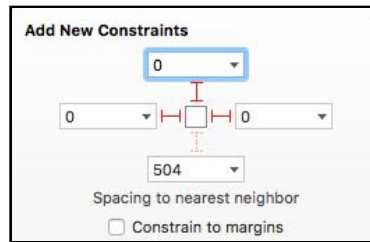
Make sure the Search Bar is not placed inside the table view. It should sit on the same level as the table view in the outline pane:



Search Bar must be below of Table View (left), not inside (right)

If you did put the Search Bar inside the Table View, you can pick it up in the outline pane and drag it below the Table View.

► Pin the Search Bar to the top and left and right edges, 3 constraints in total.

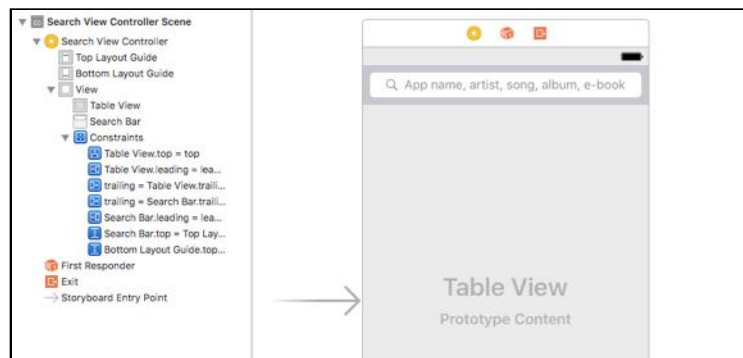


The constraints for the Search Bar

You don't need to pin the bottom of the Search Bar or give it a height constraint. Search Bars have an *intrinsic* height of 44 points.

► In the **Attributes inspector** for the Search Bar, change the **Placeholder** text to **App name, artist, song, album, e-book**.

The view controller's design should look like this:



The search view controller with Search Bar and Table View

You know what's coming next: connecting the Search Bar and the Table View to outlets on the view controller.

► Add the following outlets to **SearchViewController.swift**:

```
@IBOutlet weak var searchBar: UISearchBar!
@IBOutlet weak var tableView: UITableView!
```

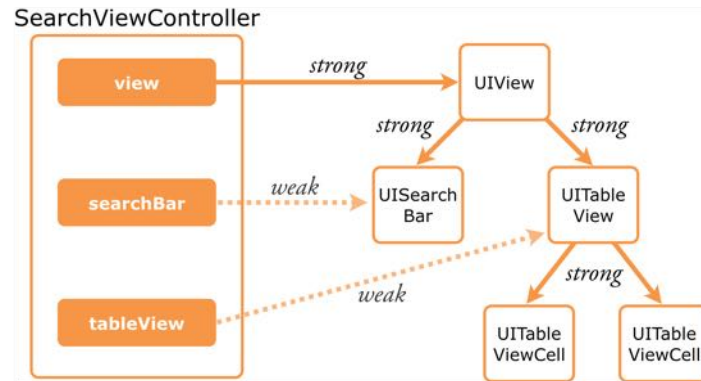
Recall that as soon as an object no longer has any strong references, it goes away – it is deallocated – and any weak references to it become `nil`.

Per Apple's recommendation you've been making your outlets weak. You may be wondering, if the references to these view objects are weak, then won't the objects get deallocated too soon?

Exercise. What is keeping these views from being deallocated? ■

Answer: Views are always part of a view hierarchy and they will always have an owner with a strong reference: their *superview*.

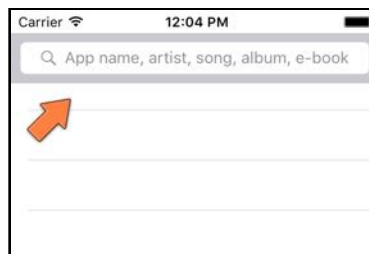
The SearchViewController's main view object holds a reference to both the search bar and the table view. This is done inside UIKit and you don't have to worry about it. As long as the view controller exists, so will these two outlets.



Outlets can be weak because the view hierarchy already has strong references

➤ Switch back to the storyboard and connect the Search Bar and the Table View to their respective outlets. (Ctrl-drag from the view controller to the object that you want to connect.)

If you run the app now, you'll notice a small problem: the first rows of the Table View are hidden beneath the Search Bar.



The first row is only partially visible

That's not so strange because you put the Search Bar on top of the table, obscuring part of the table view below.

To fix this you could nudge the Table View down a few pixels. However, according to the iOS design guidelines the content of a view controller should take up the entire screen space.

It's better to leave the size of the Table View alone and to make the Search Bar partially translucent to let the contents of the table cells shine through. But it would still be nice to see the first few rows in their entirety.

You can compensate for this with the **content inset** attributes of the Table View. Unfortunately, this attribute is unavailable in Interface Builder so you'll have to do

this from code.

- Add the following line to `viewDidLoad()` in **SearchViewController.swift**:

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    tableView.contentInset = UIEdgeInsets(top: 64, left: 0, bottom: 0,  
                                          right: 0)  
}
```

This tells the table view to add a 64-point margin at the top, made up of 20 points for the status bar and 44 points for the Search Bar.

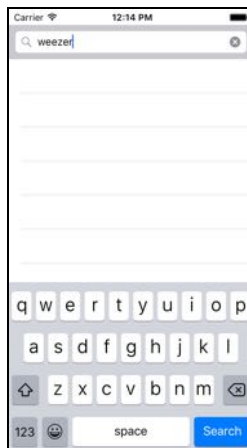
Now the first row will always be visible, and when you scroll the table view the cells still go under the search bar. Nice.

Doing fake searches

Before you search the iTunes store, it's good to understand how the `UISearchBar` component works.

In this section you'll get the text to search for from the search bar and use that to put some fake search results into the table view. Once you've got that working, you can build in the web service. Small steps!

- Run the app. If you tap in the search bar, the on-screen keyboard will appear, but it still doesn't do anything when you tap the Search button.



Keyboard with Search button

(If you're using the Simulator you may need to press **⌘K** to bring up the keyboard, and **Shift+⌘K** to allow typing from your Mac keyboard.)

Listening to the search bar is done – how else? – with a delegate. Let's put this delegate code into an extension.

- Add the following to the bottom of **SearchViewController.swift**, below the final

closing bracket:

```
extension SearchViewController: UISearchBarDelegate {  
    func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {  
        print("The search text is: '\(searchBar.text!)'")  
    }  
}
```

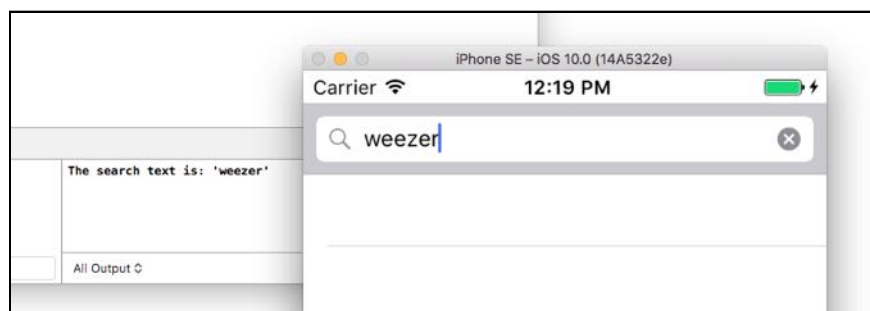
Recall that you can use extensions to organize your source code. By putting all the `UISearchBarDelegate` stuff into its own extension you keep it together in one place and out of the way of the rest of the code.

The `UISearchBarDelegate` protocol has a method `searchBarSearchButtonClicked()` that is invoked when the user taps the Search button on the keyboard. You will implement this method to put some fake data into the table.

In a little while you'll make this method send a network request to the iTunes store to find songs, movies and e-books that match the search text that the user typed, but let's not do too many new things at once!

Tip: I always put strings in between single quotes when I use `print()`. That way you can easily see whether there are any trailing or leading spaces in the string. Also note that `searchBar.text` is an optional, so we need to unwrap it. It will never actually return nil so a `!` will do just fine.

- In the storyboard, **Ctrl-drag** from the Search Bar to Search View Controller (or the yellow circle at the top). Connect to **delegate**.
- Run the app, type something in the search bar and press the Search button. The Xcode Debug pane should now print the text you typed.



The search text in the debug pane

- Add the following two (empty) extensions to **SearchViewController.swift**:

```
extension SearchViewController: UITableViewDataSource {  
}  
  
extension SearchViewController: UITableViewDelegate {
```

```
}
```

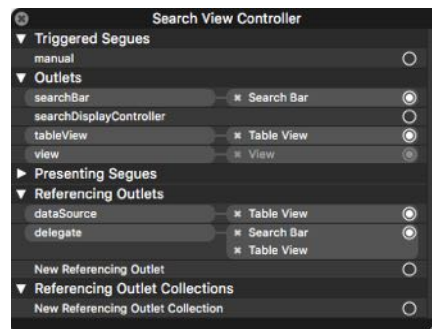
Adding the UITableViewDataSource and UITableViewDelegate protocols wasn't necessary in the previous tutorials because you used a UITableViewController there, which by design already conforms to these protocols.

For this app you're using a regular view controller and therefore you'll have to hook up the data source and delegate protocols yourself.

► In the storyboard, **Ctrl-drag** from the Table View to Search View Controller. Connect to **dataSource**. Repeat and connect to **delegate**.

Note that you connected something to Search View Controller's "delegate" twice: the Search Bar and the Table View.

The way Interface Builder presents this is a little misleading: the delegate outlet is not from SearchViewController, but belongs to the thing that you Ctrl-dragged from. So you connected the SearchViewController to the delegate outlet on the Search Bar and also to the delegate (and dataSource) outlets on the Table View:



The connections from Search View Controller to the other objects

► Build the app. Whoops... Xcode says, "Not so fast, buddy!"

By making the extension you said the SearchViewController would play the role of table view data source but you didn't actually implement any of those data source methods yet.

► Change the first extension to:

```
extension SearchViewController: UITableViewDataSource {
    func tableView(_ tableView: UITableView,
                  numberOfRowsInSection section: Int) -> Int {
        return 0
    }

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

This simply tells the table view that it has no rows yet. Soon you'll give it some fake data to display, but for now you just want to be able to run the app without errors.

Often you can declare to conform to a protocol without implementing any of its methods. This works fine for `UISearchBarDelegate` and `UITableViewDelegate`, but obviously not in the case of `UITableViewDataSource`!

A protocol can have optional and required methods. If you forget a required method, a compiler error is your reward. (Swift is more strict about this than Objective-C, which simply crashes if a required method is missing.)

► Build and run the app to make sure everything still works.

Note: Did you notice a difference between these data source methods and the ones from the previous tutorials? Look closely...

Answer: They don't have the `override` keyword in front of them.

In the previous apps, `override` was necessary because you were dealing with a subclass of `UITableViewController`, which already provides its own version of the `tableView:numberOfRowsInSection` and `cellForRowAt` methods.

In those apps you were "overriding" or replacing those methods with your own versions, hence the need for the `override` keyword.

Here, however, your base class is not a table view controller but a regular `UIViewController`. Such a view controller doesn't have any table view methods yet, so you're not overriding anything here.

As you know by now, a table view needs some kind of data model. Let's start with a simple Array.

► Add an instance variable for the array (this goes inside the class brackets, not in any of the extensions):

```
var searchResults: [String] = []
```

This creates an empty array object that can hold strings. You could also have written this as:

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

Both lines do the exact same thing. You'll see both styles of notation used in Swift programs.

The search bar delegate method will put some fake data into this array and then use it to fill up the table.

► Replace the `searchBarSearchButtonClicked()` method with:

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
    searchResults = []

    for i in 0...2 {
        searchResults.append(String(format: "Fake Result %d for '%@'", i,
                                         searchBar.text!))
    }

    tableView.reloadData()
}
```

Here the notation `[]` means you instantiate a new `[String]` array and put it into the `searchResults` instance variable. This is done each time the user performs a search. If there was already a previous array then it is thrown away and deallocated. (You could also have written `searchResults = [String]()` to do the same thing.)

You add a string with some text into the array. Just for fun, that is repeated 3 times so your data model will have three rows in it.

When you write `for i in 0...2`, it creates a loop that repeats three times because the *closed range* `0...2` contains the numbers 0, 1, and 2. Note that this is different from the *half-open range* `0..<2`, which only contains 0 and 1. You could also have written `1...3` but programmers like to start counting at 0.

You've seen format strings before. The format specifier `%d` is a placeholder for integer numbers. Likewise, `%f` is for numbers with a decimal point (the floating-point numbers). The placeholder `%@` is for all other kinds of objects, such as strings.

The last statement in the method reloads the table view to make the new rows visible, which means you have to adapt the data source methods to read from this array as well.

► Change the `tableView(numberOfRowsInSection)` method to:

```
func tableView(_ tableView: UITableView,
               numberOfRowsInSection section: Int) -> Int {
    return searchResults.count
}
```

This simply returns the number of elements in the `searchResults` array. When the app first starts up, `searchResults` will have an empty array because no search is done yet and simply returns 0.

► Finally, change `tableView(cellForRowAt)` to:

```
func tableView(_ tableView: UITableView,
               cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cellIdentifier = "SearchResultCell"

    var cell: UITableViewCell! = tableView.dequeueReusableCell(
                                                withIdentifier: cellIdentifier)

    if cell == nil {
        cell = UITableViewCell(style: .default,
```

```

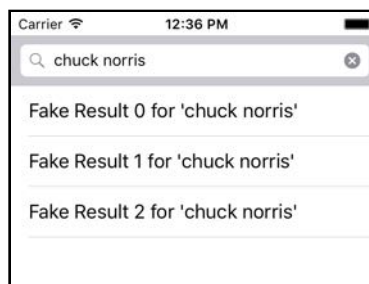
        reuseIdentifier: cellIdentifier)
    }
    cell.textLabel!.text = searchResults[indexPath.row]
    return cell
}

```

You've seen something like this before. You create a `UITableViewCell` by hand and put the data for this row into its text label.

► Run the app. If you search for anything, a couple of fake results get added to the data model and are shown in the table.

Search for something else and the table view updates with new fake results.



The app shows fake results when you search

There are some improvements you can make. To begin with, it's not very nice that the keyboard stays on the screen after you press the Search button. It obscures about half of the table view and there is no way to dismiss the keyboard by hand.

► Add the following line at the start of `searchBarSearchButtonClicked()`:

```

func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
    searchBar.resignFirstResponder()
    . . .
}

```

This tells the `UISearchBar` that it should no longer listen to keyboard input. As a result, the keyboard will hide itself until you tap inside the search bar again.

You can also configure the table view to dismiss the keyboard with a gesture.

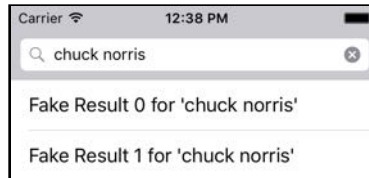
► In the storyboard, select the Table View. Go to the **Attributes inspector** and set **Keyboard** to **Dismiss interactively**.

The search bar still has an ugly white gap above it. It would look a lot better if the status bar area was unified with the search bar. Recall that the navigation bar in the Map screen from the `MyLocations` tutorial had a similar problem. You can use the same trick to fix it.

► Add the following method to the `SearchBarDelegate` extension:

```
func position(for bar: UIBarPositioning) -> UIBarPosition {
    return .topAttached
}
```

Now the app looks a lot smarter:



The search bar is "attached" to the top of the screen

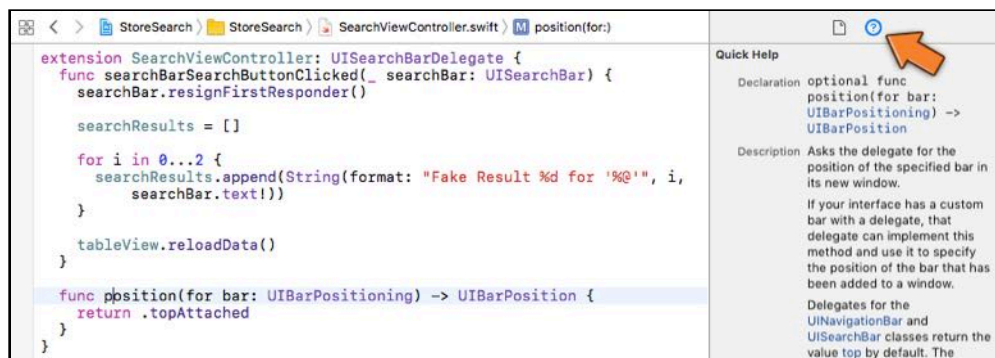
If you were to look in the API documentation for `UISearchBarDelegate` you wouldn't find this `position(for)` method. Instead, it is part of the `UIBarPositioningDelegate` protocol, which the `UISearchBarDelegate` protocol extends. (Like classes, protocols can inherit from other protocols.)



The API documentation

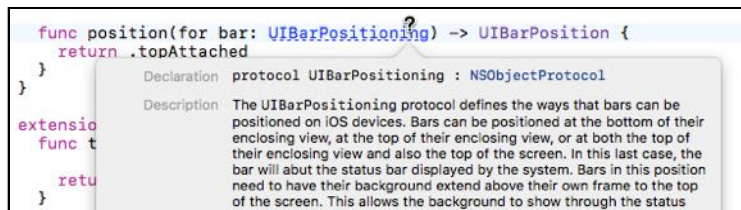
Xcode comes with a big library of documentation for developing iOS apps. Basically everything you need to know is in here. Learn to use the Xcode documentation browser – it will become your best friend!

There are a few ways to get documentation about a certain item in Xcode. There is Quick Help, which shows info about the thing under the text cursor:

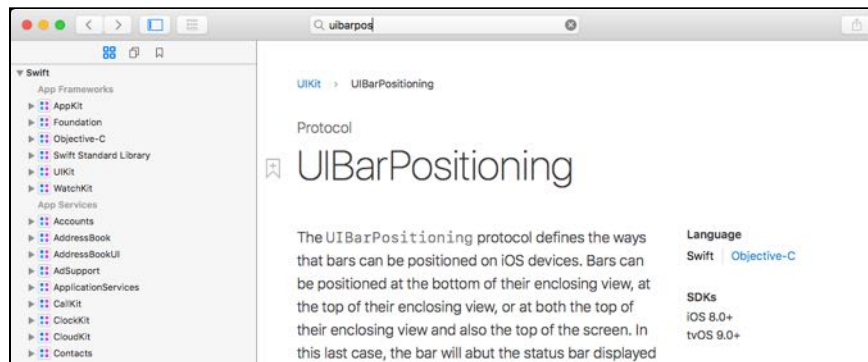


Simply have the **Quick Help inspector** open (the second tab in the inspector pane) and it will show context-sensitive help. Put the text cursor on the thing you want to know more about and the inspector will give a summary on it. You can click any of the blue text to jump to the full documentation.

You can also get pop-up help. Hold down the **Option** (Alt) key and hover over the item that you want to learn more about. Then click the mouse:



And of course, there is the full-fledged documentation window. You can access it from the **Help** menu, under **Documentation and API Reference**. Use the bar at the top to search for the item that you want to know more about:



Improving the data model

So far you've added `String` objects to the `searchResults` array, but that's a bit limited. The search results that you'll get back from the iTunes store include the product name, the name of the artist, a link to an image, the purchase price, and much more.

You can't fit all of that in a single string, so let's create a new class to hold this data.

➤ Add a new file to the project using the **Swift File** template. Name the new class **SearchResult**.

➤ Replace the contents of **SearchResult.swift** with:

```
class SearchResult {
    var name = ""
    var artistName = ""
```



```
}
```

This adds two properties to the new `SearchResult` class. In a little while you'll add several others.

In the `SearchViewController` you will no longer add strings to the `searchResults` array, but instances of `SearchResult`.

► In **`SearchViewController.swift`**, change the `for in` loop in the search bar delegate method to:

```
for i in 0...2 {  
    let searchResult = SearchResult()  
    searchResult.name = String(format: "Fake Result %d for", i)  
    searchResult.artistName = searchBar.text!  
    searchResults.append(searchResult)  
}
```

This creates the new `SearchResult` object and simply puts some fake text into its `name` and `artistName` properties. Again, you do this in a loop because just having one search result by itself is a bit lonely.

Exercise. At this point Xcode gives an error message. Can you explain why? ■

Answer: The type of `searchResults` is `array-of-String`, but here you're trying to put `SearchResult` objects into the array. To make it accept `SearchResult` objects, you also need to change the declaration of the instance variable:

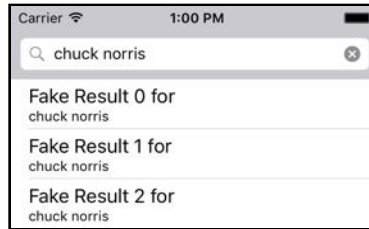
```
var searchResults: [SearchResult] = []
```

► At this point, `tableView(cellForRowAt)` still expects the array to contain strings so also update that method:

```
func tableView(_ tableView: UITableView,  
               cellForRowAt indexPath: IndexPath) -> UITableViewCell {  
    . . .  
    if cell == nil {  
        cell = UITableViewCell(style: .subtitle,           // change  
                               reuseIdentifier: cellIdentifier)  
    }  
  
    let searchResult = searchResults[indexPath.row]  
    cell.textLabel!.text = searchResult.name  
    cell.detailTextLabel!.text = searchResult.artistName  
    return cell  
}
```

Instead of a regular table view cell this now uses a “subtitle” cell style. You put the contents of the `artistName` property into the detail (subtitle) text label.

► Run the app; it should look like this:



Fake results in a subtitle cell

Nothing found

When you add searching capability to your apps, you'll usually have to handle the following situations:

1. The user did not perform a search yet.
2. The user performed the search and received one or more results. That's what happens in the current version of the app: for every search you'll get back a handful of `SearchResult` objects.
3. The user performed the search and there were no results. It's usually a good idea to explicitly tell the user there were no results. If you display nothing at all the user may wonder whether the search was actually performed or not.

Even though the app doesn't do any actual searching yet – everything is fake – there is no reason why you cannot fake the latter situation as well.

For the sake of good taste, the app will return 0 results when the user searches for "justin bieber", just so you know the app can handle this kind of situation.

► In `searchBarSearchButtonClicked()`, put the following if-statement around the `for in` loop:

```
if searchBar.text! != "justin bieber" {  
    for i in 0...2 {  
        . . .  
    }  
}
```

The change here is pretty simple. You have added an if-statement that compares the search text to "justin bieber". Only if there is no match will this create the `SearchResult` objects and add them to the array.

► Run the app and do a search for "justin bieber" (all lowercase). The table should stay empty.

You can improve the user experience by showing the text "(Nothing found)" instead, so the user knows beyond a doubt that there were no search results.

► Change the bottom part of `tableView(cellForRowAt)` to:

```
if searchResults.count == 0 {  
    cell.textLabel!.text = "(Nothing found)"  
    cell.detailTextLabel!.text = ""  
} else {  
    let searchResult = searchResults[indexPath.row]  
    cell.textLabel!.text = searchResult.name  
    cell.detailTextLabel!.text = searchResult.artistName  
}  
return cell
```

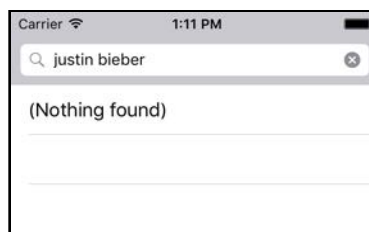
That alone is not enough. When there is nothing in the array, `searchResults.count` is 0, right? But that also means the data source's `numberOfRowsInSection` will return 0 and the table view stays empty – this “Nothing found” row will never show up.

► Change `tableView(numberOfRowsInSection)` to:

```
func tableView(_ tableView: UITableView,  
               numberOfRowsInSection section: Int) -> Int {  
    if searchResults.count == 0 {  
        return 1  
    } else {  
        return searchResults.count  
    }  
}
```

If there are no results this returns 1, for the row with the text “(Nothing Found)”. This works because both `numberOfRowsInSection` and `cellForRowAt` check for this special situation.

► Try it out:



One can hope...

Unfortunately, the text “Nothing found” also appears when the user did not actually search for anything yet. That’s a little silly.

The problem is that you have no way to distinguish between “not searched yet” and “nothing found”. Right now, you can only tell whether the `searchResults` array is empty but not what caused this.

Exercise. How would you solve this little problem? ■

There are two obvious solutions that come to mind:

- Make `searchResults` into an optional. If it is `nil`, i.e. it has no value, then the

user hasn't searched yet. That's different from the case where the user did search and no matches were found.

- Use a separate boolean variable to keep track of this.

It may be tempting to choose the optional, but it's best to avoid optionals if you can. They make the logic more complex, they can cause the app to crash if you don't unwrap them properly, and they require `if let` statements everywhere. Optionals certainly have their uses but here they are not really necessary.

So we'll opt for the boolean. (But feel free to come back and try the optional as well, and compare the differences. It'll be a great exercise!)

- Add the new instance variable:

```
var hasSearched = false
```

- In the search bar delegate method, set this variable to `true`. It doesn't really matter where you do this, as long as it happens before the table view is reloaded.

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {  
    .  
    .  
    hasSearched = true  
    tableView.reloadData()  
}
```

- And finally, change `tableView(numberOfRowsInSection)` to look at the value of this new variable:

```
func tableView(_ tableView: UITableView,  
               numberOfRowsInSection section: Int) -> Int {  
    if !hasSearched {  
        return 0  
    } else if searchResults.count == 0 {  
        return 1  
    } else {  
        return searchResults.count  
    }  
}
```

Now the table view remains empty until you first search for something. Try it out! (Later on in the tutorial you'll see a much better way to handle this using an enum – and it will blow your mind!)

One more thing, if you currently tap on a row it will become selected and stays selected.

- To fix that, add the following methods inside the `UITableViewDelegate` extension:

```
func tableView(_ tableView: UITableView,  
               didSelectRowAt indexPath: IndexPath) {  
    tableView.deselectRow(at: indexPath, animated: true)  
}
```

```
func tableView(_ tableView: UITableView,
               willSelectRowAt indexPath: IndexPath) -> IndexPath? {
    if searchResults.count == 0 {
        return nil
    } else {
        return indexPath
    }
}
```

The `tableView(didSelectRowAt)` method will simply deselect the row with an animation, while `willSelectRowAt` makes sure that you can only select rows with actual search results.

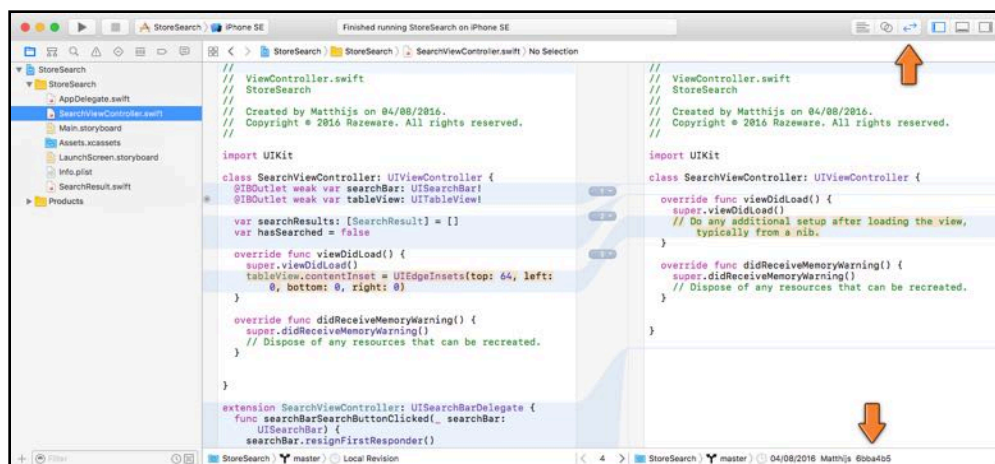
If you tap on the (Nothing Found) row now you will notice that it does not turn gray at all. (Actually, the row may still turn gray if you press down on it for a short while. That happens because you did not change the `selectionStyle` property of the cell. You'll fix that in the next section.)

➤ This is a good time to commit the app. Go to **Source Control** → **Commit** (or press the **⌘+Option+C** keyboard shortcut).

Make sure all the files are selected, review your changes, and type a good commit message – something like “Add a search bar and table view. The search puts fake results in the table for now.” Press the **Commit** button to finish.

Note: It is customary to write commit messages in the present tense. That's why I wrote “Add a search bar” instead of “Added a search bar”.

If you ever want to look back through your commit history, you can either do that from the **Source Control** → **History** window or from the **Version editor**, pictured below:



Viewing revisions in the Version editor

You switch to the Version editor with the button in the toolbar at the top of the Xcode window.

In the screenshot above, the current version is shown on the left and the previous version on the right. You can switch between versions with the bar at the bottom. The Version editor is a very handy tool for viewing the history of changes in your source files.

The app isn't very impressive yet but you've laid the foundation for what is to come. You have a search bar and know how to take action when the user presses the Search button. The app also has a simple data model that consists of an array with `SearchResult` objects, and it can display these search results in a table view.

You can find the project files for the first part of this app under **01 - Search Bar** in the tutorial's Source Code folder.

Before you make the app do a real search on the iTunes store, first let's make the table view look a little better. Appearance does matter!

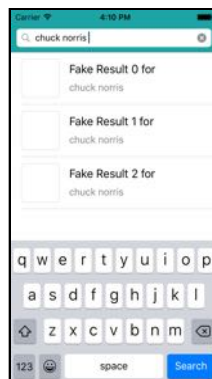
Custom table cells and nibs

In the previous tutorials you used prototype cells to create your own table view cell layouts. That works great but there's another way: in this tutorial you'll create a "nib" file with the design for the cell and load your table view cells from that. The principle is very similar to prototype cells.

A nib, also called a xib, is very much like a storyboard except that it only contains the design for a single thing. That thing can be a view controller but it can also be an individual view or table view cell. A nib is really nothing more than a container for a "freeze dried" object that you can edit in Interface Builder.

In practice, many apps consist of a combination of nibs and storyboard files, so it's good to know how to work with both.

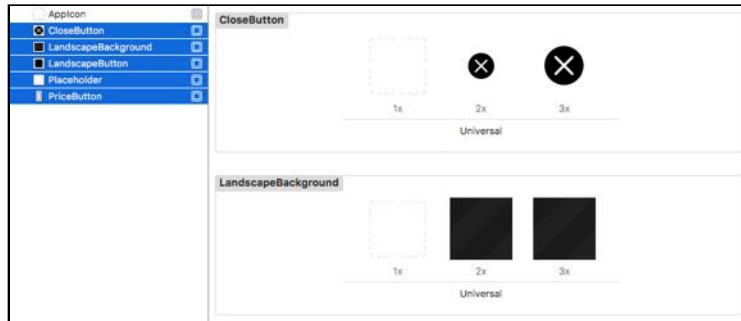
This is what you're going to make in this section:



The app with better looks

The app still uses the same fake data, but you'll make it look a bit better.

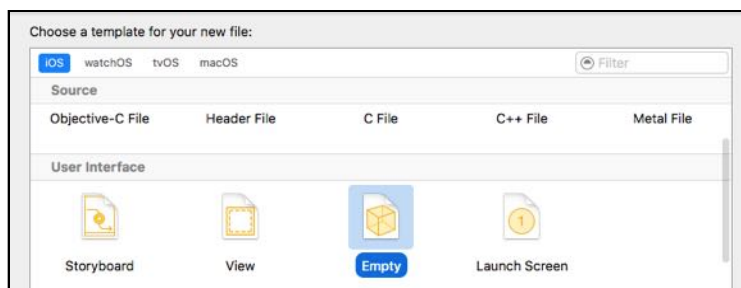
► First, add the contents of the **Images** folder from this tutorial's resources into the project's asset catalog, **Assets.xcassets**.



Imported images in the asset catalog

Each of the images comes in two versions: 2x and 3x. There are no low-resolution 1x devices that can run iOS 10. Even though you'll also make this app run on iPads, iOS 10 only supports iPads with a 2x Retina screen. So there's no point in including 1x images.

► Add a new file to the project. Choose the **Empty** template from the **User Interface** category (scroll down in the template chooser). This will create a new nib without anything in it.



Adding an empty nib to the project

► Click **Next** and save the new file as **SearchResultCell**.

This adds a nib with no contents to the project. Open **SearchResultCell.xib** and you will see an empty canvas.

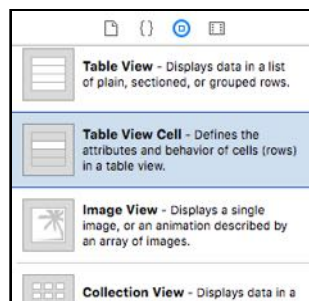
Xib or nib

I've been calling it a nib but the file extension is **.xib**. So what is the difference? In practice these terms are used interchangeably. Technically speaking, a xib file is compiled into a nib file that is put into your application bundle. The term nib mostly stuck for historical reasons (it stands for *NeXT*

Interface Builder, after the old NeXT computers from the 1990s).

You can consider the terms “xib file” and “nib file” to be equivalent. The preferred term seems to be nib, so that is what I will be using from now on. (This won’t be the last time computer terminology is confusing, ambiguous or inconsistent. The world of programming is full of colorful slang.)

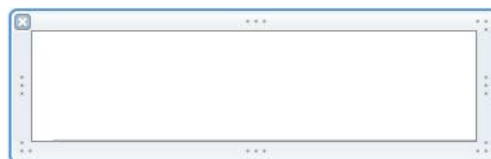
- Use the **View as:** panel to switch to **iPhone SE** dimensions. As usual, we’ll design for this device first and then use Auto Layout to make the user interface adapt to the larger iPhone models.
- From the Object Library, drag a new **Table View Cell** into the canvas:



The Table View Cell in the Object Library

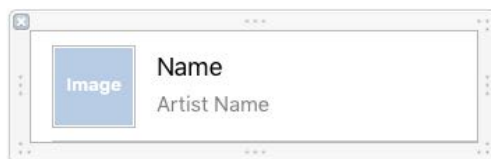
- Select the new Table View Cell and go to the **Size inspector**. Type 80 in the **Height** field (not Row Height). Make sure **Width** is 320, the width of the iPhone SE screen.

The cell now looks like this:



An empty table view cell

- Drag an **Image View** and two **Labels** into the cell, like this:



The design of the cell

- The Image View is positioned at X:15, Y:10, Width:60, Height:60.
- The **Name** label is at X:90, Y:15, Width:222, Height:22. Its font is **System 18**.
- The **Artist Name** label is at X:90, Y:44, Width:222, Height:18. Font is **System 15** and Color is black with 50% opacity.

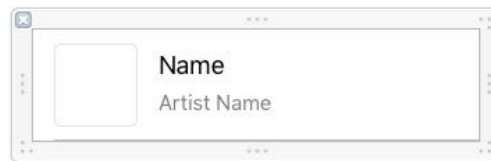
As you can see, editing a nib is just like editing a storyboard. The difference is that the canvas is a lot smaller, but that's because you're only editing a single table view cell, not an entire view controller.

- The Table View Cell itself needs to have a reuse identifier. You can set this in the **Attributes inspector** to the value **SearchResultCell**.

The image view will hold the artwork for the found item, such as an album cover, book cover, or an app icon. It may take a few seconds for these images to be loaded, so until then it's a good idea to show a placeholder image. That placeholder is part of the image files you just added to the project.

- Select the Image View. In the **Attributes inspector**, set **Image** to **Placeholder**.

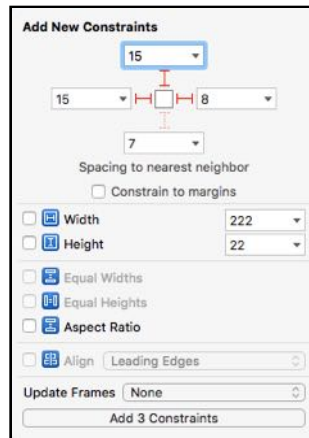
The cell design now looks like this:



The cell design with placeholder image

You're not done yet. The design for the cell is only 320 points wide but the iPhone 6s, 7, and Plus are wider than that. The cell itself will resize to accommodate those larger screens but the labels won't, potentially causing their text to be cut off. You'll have to add some Auto Layout constraints to make the labels resize along with the cell.

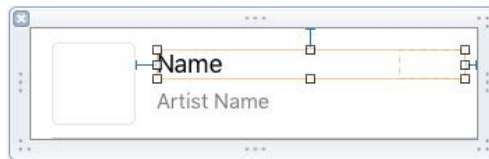
- Select the **Name** label and open the **Pin menu**. Uncheck **Constrain to margins** and select the **top**, **left**, and **right** pins (but not the bottom one):



The constraints for the Name label

This time, leave **Update Frames** set to **None**. When enabled, the Update Frames option will move and resize the label according to the constraints you've set on it. That will not do the correct thing in this case (if you're curious, try it out and see what happens).

➤ Click **Add 3 Constraints** to finish. The nib now looks like this:



The Name label has insufficient constraints

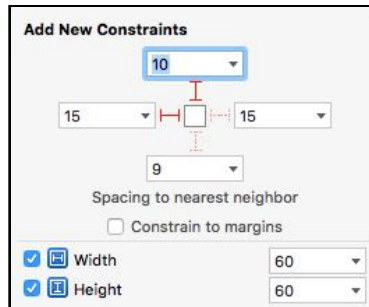
The orange rectangle indicates that something's not right with the constraints from the Name label. Apparently the width of the label is incorrect; the dotted box over on the right is what Auto Layout thinks should be the size and position for the label according to the constraints.

Note: If you had set Update Frames to "Items of New Constraints", then Interface Builder would have moved the label to where the dotted box is. That's why you left it set to None because you didn't want it to do that here.

Auto Layout is entitled to its opinion, of course, but over in the corner is not where *you* want the label to be. Its current position is just fine, so you'll have to add a couple more constraints to tell Auto Layout that this is really what you intended.

The solution is to pin the Image View. Remember that each view always needs to have enough constraints to uniquely determine its position and size. The Name label is connected to the Image View on the left, but the Image View doesn't have any constraints of its own.

► Select the **Image View** and pin it to the **top** and **left** sides of the cell. Also give it **Width** and **Height** constraints so that its size is always fixed to 60 by 60 points:



The constraints for the Image View

The orange box from the Name label should have disappeared. If you select this label, it should show three blue bars and nothing in orange.

► Finally, pin the **Artist Name** label to the **left**, **right**, and **bottom**.

That concludes the design for this cell. Now you have to tell the app to use this nib.

► In **SearchViewController.swift**, add these lines to the bottom of `viewDidLoad()`:

```
let cellNib = UINib(nibName: "SearchResultCell", bundle: nil)
tableView.register(cellNib, forCellReuseIdentifier: "SearchResultCell")
```

The `UINib` class is used to load nibs. Here you tell it to load the nib you just created (note that you don't specify the `.xib` file extension). Then you ask the table view to register this nib for the reuse identifier "SearchResultCell".

From now on, when you call `dequeueReusableCell(withIdentifier)` for the identifier "SearchResultCell", `UITableView` will automatically make a new cell from the nib – or reuse an existing cell if one is available, of course. And that's all you need to do.

► Change `tableView(cellForRowAt)` to:

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

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

    if searchResults.count == 0 {
        . . .
    } else {
        . . .
    }
    return cell
}
```

You were able to replace this chunk of code,

```
let cellIdentifier = "SearchResultCell"

var cell: UITableViewCell! = tableView.dequeueReusableCell(
    withIdentifier: cellIdentifier)

if cell == nil {
    cell = UITableViewCell(style: .subtitle,
        reuseIdentifier: cellIdentifier)
}
```

with just one statement. It's almost exactly like using prototype cells, except that you have to create your own nib object and you need to register it with the table view beforehand.

Note: The call to `dequeueReusableCell(withIdentifier)` now takes a second parameter, `for:`, that takes an `IndexPath` value. This variant of the `dequeueReusableCell` method lets the table view be a bit smarter, but it only works when you have registered a nib with the table view (or when you use a prototype cell).

► Run the app and do a (fake) search. Yikes, the app crashes.

Exercise. Any ideas why? ■

Answer: Because you made your own cell design, you should no longer use the `textLabel` and `detailTextLabel` properties of `UITableViewCell`.

Every table view cell – even a custom cell that you load from a nib – has a few labels and an image view of its own, but you should only employ these when you're using one of the standard cell styles: `.default`, `.subtitle`, etc. If you use them on custom cells then these labels get in the way of your own labels.

So here you shouldn't use `textLabel` and `detailTextLabel` to put text into the cell, but make your own properties for your own labels.

Where do you put these properties? In a new class, of course. You're going to make a new class named `SearchResultCell` that extends `UITableViewCell` and that has properties (and logic) for displaying the search results in this app.

► Add a new file to the project using the **Cocoa Touch Class** template. Name it **SearchResultCell** and make it a subclass of **UITableViewCell**. ("Also create XIB file" should be unchecked as you already have one.)

This creates the Swift file to accompany the nib file you created earlier.

► Open **SearchResultCell.xib** and select the Table View Cell. (Make sure you select the actual Table View Cell object, not its Content View.)

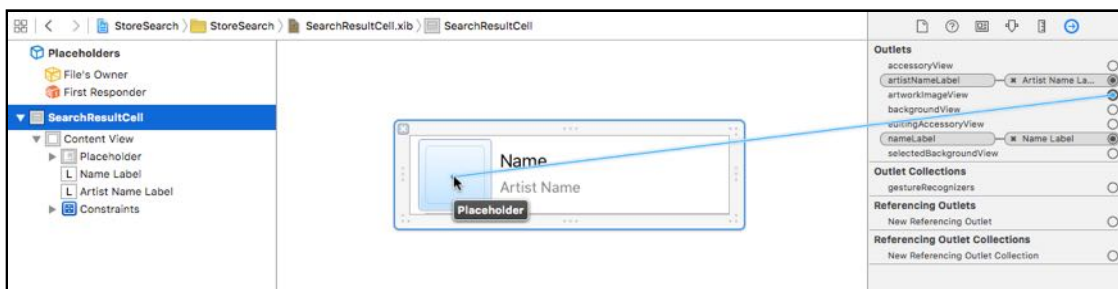
► In the **Identity inspector**, change its class from "UITableViewCell" to **SearchResultCell**.

You do this to tell the nib that the top-level view object it contains is no longer a `UITableViewCell` but your own `SearchResultCell` subclass. From now on, whenever you call `dequeueReusableCell(...)`, the table view will return an object of type `SearchResultCell`.

➤ Add the following outlet properties to **SearchResultCell.swift**:

```
@IBOutlet weak var nameLabel: UILabel!
@IBOutlet weak var artistNameLabel: UILabel!
@IBOutlet weak var artworkImageView: UIImageView!
```

➤ Hook these outlets up to the respective labels and image view in the nib. It is easiest to do this from the Connections inspector for `SearchResultCell`:



Connect the labels and image view to Search Result Cell

You can also open the Assistant editor and Ctrl-drag from the labels and image view to their respective outlet definitions. (If you've used nib files before you might be tempted to connect the outlets to File's Owner but that won't work in this case; they must be connected to the table view cell.)

Now that this is all set up, you can tell the `SearchViewController` to use these new `SearchResultCell` objects.

➤ In **SearchViewController.swift**, change "cellForRowAt" to:

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

    let cell = tableView.dequeueReusableCell(withIdentifier:
        "SearchResultCell", for: indexPath) as! SearchResultCell

    if searchResults.count == 0 {
        cell.nameLabel.text = "(Nothing found)"
        cell.artistNameLabel.text = ""
    } else {
        let searchResult = searchResults[indexPath.row]
        cell.nameLabel.text = searchResult.name
        cell.artistNameLabel.text = searchResult.artistName
    }
    return cell
}
```

Notice the change in the first line. Previously this returned a `UITableViewCell` object but now that you've changed the class name in the nib, you're guaranteed to always receive a `SearchResultCell`. (You still need to cast it with `as!`, though.)

Given that cell, you can put the name and artist name from the search result into the proper labels. You're now using the cell's `nameLabel` and `artistNameLabel` outlets instead of `textLabel` and `detailTextLabel`. You also no longer need to write `!` to unwrap because the outlets are implicitly unwrapped optionals, not true optionals.

➤ Run the app and... Hmm, that doesn't look too good:



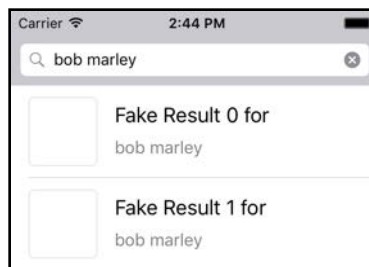
Uh oh...

The problem is that these rows aren't 80 points high. The table view isn't smart enough to figure out that these custom cells need to be higher. Fortunately this is easily fixed.

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

```
tableView.rowHeight = 80
```

➤ Run the app again and it should look something like this:



Much better!

There are a few more things to improve. Notice that you've been using the string literal `"SearchResultCell"` in a few different places? It's generally better to create a constant for such occasions.

Suppose you – or one of your co-workers – renamed the reuse identifier in one place (for whatever reason). Then you'd also have to remember to change it in all the other places where `"SearchResultCell"` is used.

It's better to limit those changes to one single spot by using a symbolic name

instead.

- Add the following to **SearchViewController.swift**, somewhere inside the class:

```
struct TableViewCellIdentifiers {  
    static let searchResultCell = "SearchResultCell"  
}
```

This defines a new struct, `TableViewCellIdentifiers`, containing a constant named `searchResultCell` with the value `"SearchResultCell"`.

Should you want to change this value, then you only have to do it here and any code that uses `TableViewCellIdentifiers.searchResultCell` will be automatically updated.

There is another reason for using a symbolic name rather than the actual value: it gives extra meaning. Just seeing the text `"SearchResultCell"` says less about its intended purpose than the symbol `TableViewCellIdentifiers.searchResultCell`.

Note: Putting symbolic constants as `static let` members inside a struct is a common trick in Swift. A static value can be used without an instance so you don't need to instantiate `TableViewCellIdentifiers` before you can use it (like you would need to do with a class).

It's allowed in Swift to place a struct *inside* a class, which permits different classes to all have their own struct `TableViewCellIdentifiers`. This wouldn't work if you placed the struct outside the class – then you'd have more than one struct with the same name in the global namespace, which is not allowed.

- Anywhere else in **SearchViewController.swift**, replace the string `"SearchResultCell"` with `TableViewCellIdentifiers.searchResultCell`.

For example, `viewDidLoad()` will now look like this:

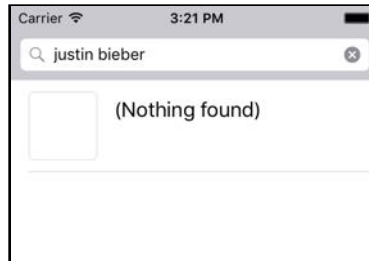
```
override func viewDidLoad() {  
    super.viewDidLoad()  
    tableView.contentInset = UIEdgeInsets(top: 64, left: 0, bottom: 0,  
                                          right: 0)  
    let cellNib = UINib(nibName:  
                        TableViewCellIdentifiers.searchResultCell, bundle: nil)  
    tableView.register(cellNib, forCellReuseIdentifier:  
                      TableViewCellIdentifiers.searchResultCell)  
    tableView.rowHeight = 80  
}
```

The other change is in `tableView(cellForRowAt)`.

- Run the app to make sure everything still works.

A new “Nothing Found” cell

Remember our friend Justin Bieber? Searching for him now looks like this:

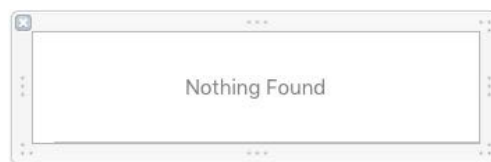


The Nothing Found label now draws like this

That’s not very pretty. It will be nicer if you gave this its own cell. That’s not too hard: you can simply make another nib for it.

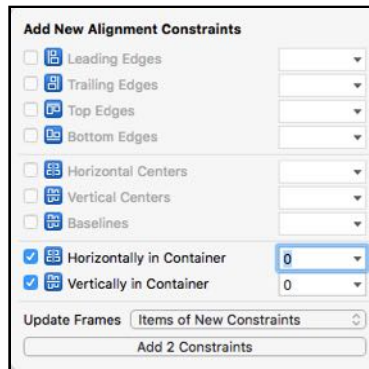
- Add another nib file to the project. Again this will be an **Empty** nib. Name it **NothingFoundCell.xib**.
- Drag a new **Table View Cell** into the canvas. Set its **Width** to 320, its **Height** to 80 and give it the reuse identifier **NothingFoundCell**.
- Drag a **Label** into the cell and give it the text **Nothing Found**. Make the text color 50% opaque black and the font **System 15**.
- Use **Editor** → **Size to Fit Content** to make the label fit the text exactly (you may have to deselect and select the label again to enable this menu option).
- Center the label in the cell, using the blue guides to snap it exactly to the center.

It should look like this:



Design of the Nothing Found cell

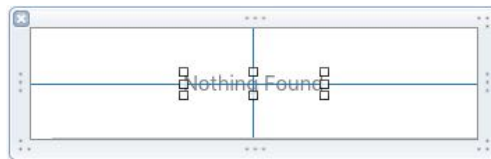
In order to keep the text centered on all devices, select the label and open the **Align menu**:



Creating the alignment constraints

► Choose **Horizontally in Container** and **Vertically in Container**. Set Update Frames to **Items of New Constraints**.

The constraints should look like this:



The constraints for the label

One more thing to fix. Remember that in “willSelectRowAt” you return nil if there are no search results to prevent the row from being selected? Well, if you are persistent enough you can still make the row appear gray as if it were selected.

For some reason, UIKit draws the selected background if you press down on the cell for long enough, even though this doesn’t count as a real selection. To prevent this, you have to tell the cell not to use a selection color.

► Select the cell itself. In the **Attributes inspector**, set **Selection** to **None**. Now tapping or holding down on the Nothing Found row will no longer show any sort of selection.

You don’t have to make a UITableViewCell subclass for this cell because there is no text to change or properties to set. All you need to do is register this nib with the table view.

► Add to the struct in **SearchViewController.swift**:

```
struct TableViewCellIdentifiers {
    static let searchResultCell = "SearchResultCell"
    static let nothingFoundCell = "NothingFoundCell"
}
```

► Add these lines to viewDidLoad(), below the other code that registers the nib:

```
cellNib = UINib(nibName: TableViewCellIdentifiers.nothingFoundCell,
                bundle: nil)
tableView.register(cellNib,
                  forCellReuseIdentifier: TableViewCellIdentifiers.nothingFoundCell)
```

This also requires you to change “let cellNib” into var because you’re re-using the cellNib local variable.

► And finally, change tableView(cellForRowAt) to:

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

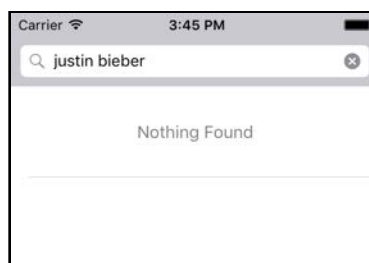
    if searchResults.count == 0 {
        return tableView.dequeueReusableCell(
            withIdentifier: TableViewCellIdentifiers.nothingFoundCell,
            for: indexPath)

    } else {
        let cell = tableView.dequeueReusableCell(
            withIdentifier: TableViewCellIdentifiers.searchResultCell,
            for: indexPath) as! SearchResultCell

        let searchResult = searchResults[indexPath.row]
        cell.nameLabel.text = searchResult.name
        cell.artistNameLabel.text = searchResult.artistName
        return cell
    }
}
```

The logic here has been restructured a little. You only make a SearchResultCell if there are actually any results. If the array is empty, you’ll simply dequeue the cell for the nothingFoundCell identifier and return it. There is nothing to configure for that cell so this one-liner will do.

► Run the app. The search results for Justin Bieber now look like this:



The new Nothing Found cell in action

Also try it out on the larger iPhones. The label should always be centered in the cell.

Sweet. It has been a while since your last commit, so this seems like a good time to secure your work.

► Commit the changes to the repository. I used the message “Use custom cells for

search results.”

Changing the look of the app

As I write this, it’s gray and rainy outside. The app itself also looks quite gray and dull. Let’s cheer it up a little by giving it more vibrant colors.

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

```
func customizeAppearance() {  
    let barTintColor = UIColor(red: 20/255, green: 160/255, blue: 160/255,  
                               alpha: 1)  
    UISearchBar.appearance().barTintColor = barTintColor  
  
    window!.tintColor = UIColor(red: 10/255, green: 80/255, blue: 80/255,  
                                alpha: 1)  
}
```

This changes the appearance of the UISearchBar – in fact, it changes *all* search bars in the application. You only have one, but if you had several then this changes the whole lot in one swoop.

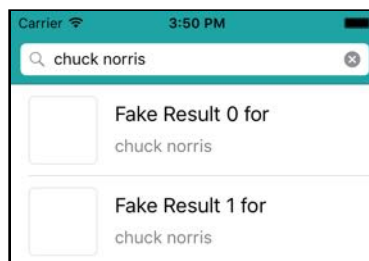
The UIColor(red, green, blue, alpha) method makes a new UIColor object based on the RGB and alpha color components that you specify.

Many painting programs let you pick RGB values going from 0 to 255 so that’s the range of color values that many programmers are accustomed to thinking in. The UIColor initializer, however, accepts values between 0.0 and 1.0, so you have to divide these numbers by 255 to scale them down to that range.

➤ Call this new method from application(didFinishLaunchingWithOptions):

```
func application(_ application: UIApplication,  
                didFinishLaunchingWithOptions launchOptions:  
                [UIApplicationLaunchOptionsKey: Any]?) -> Bool {  
    customizeAppearance()  
    return true  
}
```

➤ Run the app and notice the difference:



The search bar in the new teal-colored theme

The search bar is bluish-green, but still slightly translucent. The overall tint color is now a dark shade of green instead of the default blue. (You can currently only see the tint color in the text field's cursor but it will become more obvious later on.)



The role of App Delegate

The poor AppDelegate is often abused. People give it too many responsibilities. Really, there isn't that much for the app delegate to do.

It gets a number of callbacks about the state of the app – whether the app is about to be closed, for example – and handling those events should be its primary responsibility. The app delegate also owns the main window and the top-level view controller. Other than that, it shouldn't do much.

Some developers use the app delegate as their data model. That is just bad design. You should really have a separate class for that (or several). Others make the app delegate their main control hub. Wrong again! Put that stuff in your top-level view controller.

If you ever see the following type of thing in someone's source code, it's a pretty good indication that the application delegate is being used the wrong way:

```
let appDelegate = UIApplication.shared.delegate as! AppDelegate
appDelegate.someProperty = . . .
```

This happens when an object wants to get something from the app delegate. It works but it's not good architecture.

In my opinion, it's better to design your code the other way around: the app delegate may do a certain amount of initialization, but then it gives any data model objects to the root view controller, and hands over control. The root view controller passes these data model objects to any other controller that needs them, and so on.

This is also called *dependency injection*. I described this principle in the section "Passing around the context" in the MyLocations tutorial.



Currently, tapping a row gives it a gray selection. This doesn't go so well with the teal-colored theme so you'll give the row selection the same bluish-green tint.

That's very easy to do because all table view cells have a `selectedBackgroundView` property. The view from that property is placed on top of the cell's background, but below the other content, when the cell is selected.

➤ Add the following code to `awakeFromNib()` in **SearchResultCell.swift**:

```
override func awakeFromNib() {
    super.awakeFromNib()
    let selectedView = UIView(frame: CGRect.zero)
    selectedView.backgroundColor = UIColor(red: 20/255, green: 160/255,
                                           blue: 160/255, alpha: 0.5)
    selectedBackgroundView = selectedView
}
```

The `awakeFromNib()` method is called after this cell object has been loaded from the nib but before the cell is added to the table view. You can use this method to do additional work to prepare the object for use. That's perfect for creating the view with the selection color.

Why don't you do that in an `init` method, such as `init?(coder)?` To be fair, in this case you could. But it's worth noting that `awakeFromNib()` is called some time after `init?(coder)` and also after the objects from the nib have been connected to their outlets.

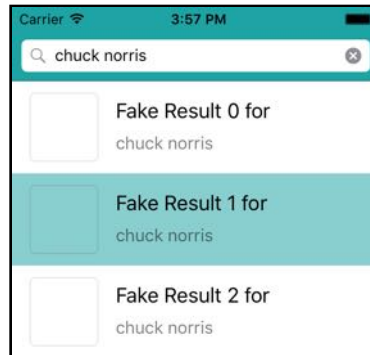
For example, in `init?(coder)` the `nameLabel` and `artistNameLabel` outlets will still be `nil` but in `awakeFromNib()` they will be properly hooked up to their `UILabel` objects. So if you wanted to do something with those outlets in code, you'd need to do that in `awakeFromNib()`, not in `init?(coder)`.

That's why `awakeFromNib()` is the ideal place for this kind of thing. (It's similar to what you use `viewDidLoad()` for in a view controller.)

Don't forget to first call `super.awakeFromNib()`, which is required. If you forget, then the superclass `UITableViewCell` – or any of the other superclasses – may not get a chance to initialize themselves.

Tip: It's always a good idea to call `super.methodName(...)` in methods that you're overriding – such as `viewDidLoad()`, `viewWillAppear()`, `awakeFromNib()`, and so on – unless the documentation says otherwise.

When you run the app, it should look like this:



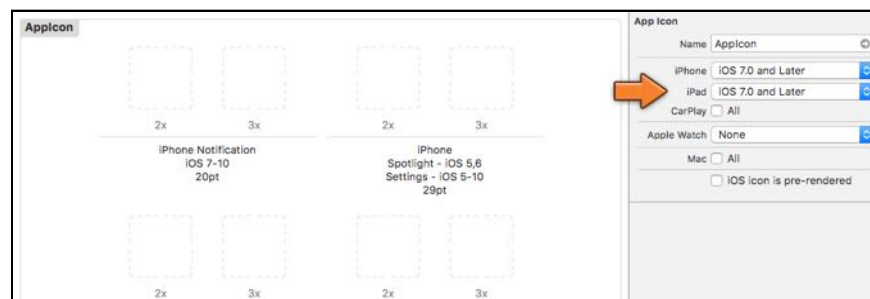
The selection color is now green

While you're at it, you might as well give the app an icon.

► Open the asset catalog (**Assets.xcassets**) and select the **AppIcon** group.

Later in this tutorial you will convert this app to run on the iPad, so you also need to add the icons for the iPad version.

► Open the **Attributes inspector** and for iPad choose **iOS 7.0 and Later**:



Enabling iPad icons

This adds nine new slots for the iPad icons.

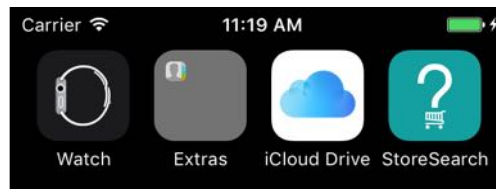
► Drag the images from the **Icon** folder from this tutorial's resources into the slots.

Keep in mind that for the 2x slots you need to use the image with twice the size in pixels. For example, you drag the **Icon-152.png** file into **iPad App 76pt, 2x**. For 3x you need to multiply the image size by 3.



All the icons in the asset catalog

- Run the app and notice that it has a nice icon:



The app icon

One final user interface tweak I'd like to make is that the keyboard will be immediately visible when you start the app so the user can start typing right away.

- Add the following line to `viewDidLoad()` in **SearchViewController.swift**:

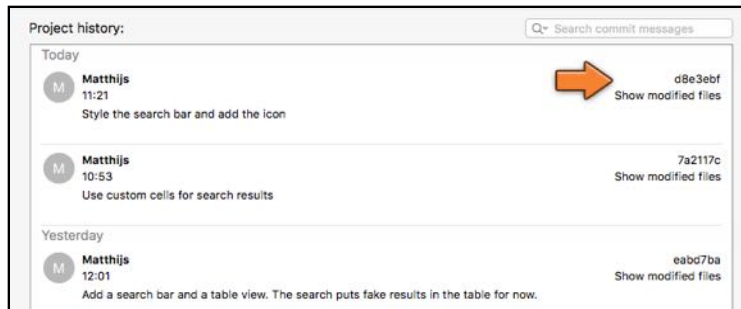
```
searchBar.becomeFirstResponder()
```

This is the inverse of `resignFirstResponder()` that you used earlier. Where "resign" got rid of the keyboard, `becomeFirstResponder()` will show the keyboard and anything you type will end up in the search bar.

- Try it out and commit your changes. You styled the search bar and added the icon.

Tagging the commits

If you look through the various commits you've made so far, you'll notice a bunch of strange numbers, such as "d8e3ebf":

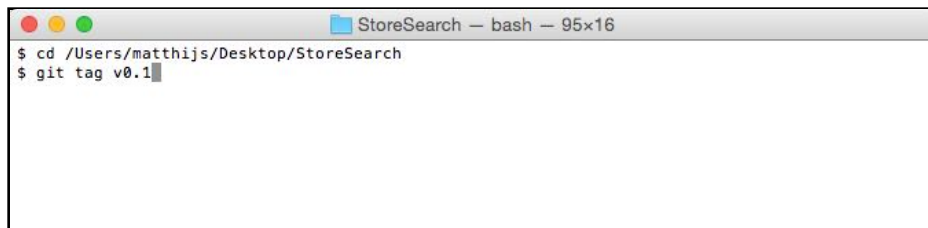


The commits are listed in the history window but have weird numbers

Those are internal numbers that Git uses to uniquely identify commits (known as the “hash”). Such numbers aren’t very nice for us humans so Git also allows you to “tag” a certain commit with a more friendly label.

Unfortunately, at the time of writing, Xcode does not support this tag command. You can do it from a Terminal window, though.

- Open the **Terminal** (from **Applications/Utilities**).
- Type “**cd** ” (with a space behind it) and from Finder drag the folder that contains the StoreSearch project into the Terminal. Then press Enter. This will make the Terminal go to your project directory.
- Type the command **git tag v0.1**

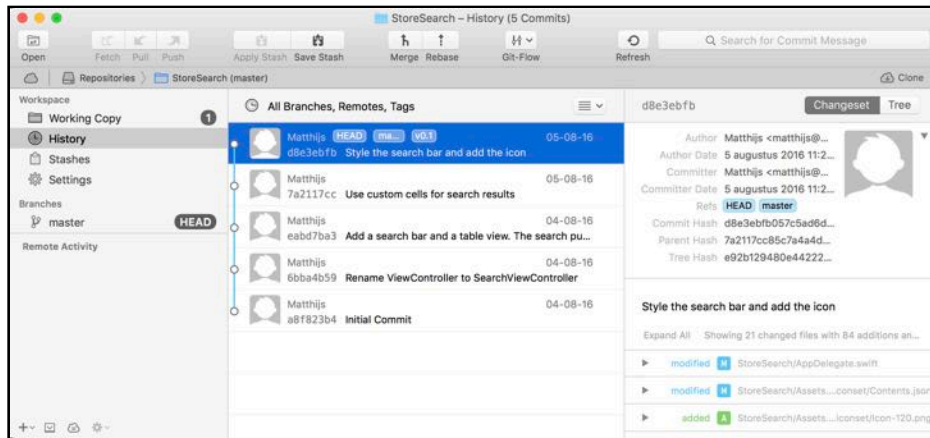


Doing git tag from the Terminal

Later you can refer to this particular commit as “v0.1”.

Note: It’s possible you may get a popup saying you first need to install the command line developer tools. Press **Install** to go ahead. If typing **git** in Terminal gives you a “command not found” error, then type the command **xcode-select --install** first.

It’s a bit of a shame that Xcode doesn’t show these Git tags, as they’re really handy, but third-party tools such as Tower do.



Viewing the Git repository with Tower

Xcode works quite well with Git but it only supports the basic features. To take full advantage of Git you'll probably need to learn how to use the Terminal or get a tool such as Tower (git-tower.com, 30-day free trial) or SourceTree (free on the Mac App Store).

You can find the project files for the app up to this point under **02 - Custom Table Cells** in the tutorial's Source Code folder.

The debugger

Xcode has a built-in **debugger**. Unfortunately, a debugger doesn't actually get the bugs out of your programs; it just lets them crash in slow motion so you can get a better idea of what is wrong.

Like a police detective, the debugger lets you dig through the evidence after the damage has been done, in order to find the scoundrel who did it.

Let's introduce a bug into the app so that it crashes. Knowing what to do when your app crashes is very important.

Thanks to the debugger, you don't have to stumble in the dark with no idea what just happened. Instead, you can use it to quickly pinpoint what went wrong and where. Once you know those two things, figuring out *why* it went wrong becomes a lot easier.

► Change **SearchViewController.swift**'s `numberOfRowsInSection` method to:

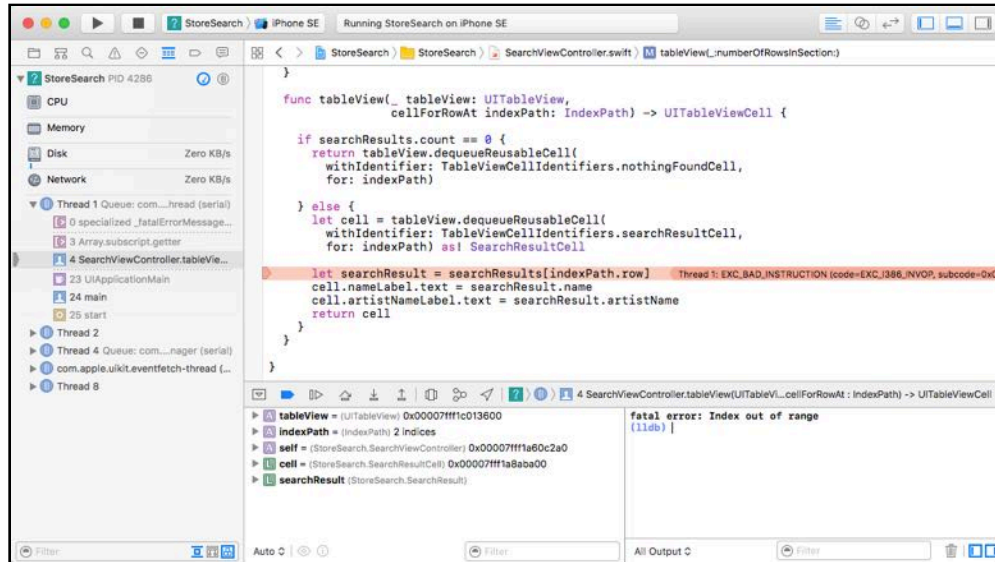
```
func tableView(_ tableView: UITableView,
               numberOfRowsInSection section: Int) -> Int {
    if !hasSearched {
        . . .
    } else if searchResults.count == 0 {
        . . .
    } else {
```

```

    }
    return searchResults.count + 1 // only this line is different
}

```

► Now run the app and search for something. The app crashes and the Xcode window changes to something like this:



The Xcode debugger appears when the app crashes

The crash is: **Thread 1: EXC_BAD_INSTRUCTION**. Sounds nasty!

There are different types of crashes, with wonderful names such as SIGABRT, EXC_BAD_ACCESS, and the one you have here, EXC_BAD_INSTRUCTION.

This is actually a pretty good crash to have – as far as that’s possible anyway. It means your app died in a controlled fashion. You did something you were not supposed to but Swift caught this and politely terminated the app with an error message.

That error message is an important clue and you can find it in Xcode’s Debug area:

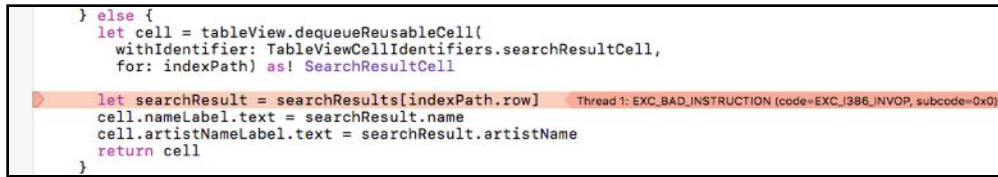
```
fatal error: Index out of range
```

According to the error message, the index that was used to access some array is larger than the number of items inside the array. In other words, the index is “out of range”. That is a common error with arrays and you’re likely to make this mistake more than once in your programming career.

Now that you know what went wrong, the big question is: *where* did it go wrong? You may have many calls to `array[index]` in your app, and you don’t want to have to dig through the entire code to find the culprit.

Thankfully, you have the debugger to help you out. In the source code editor it

already points at the offending line:



The debugger points at the line that crashed

Important: This line isn't necessarily the *cause* of the crash – after all, you didn't change anything in this method – but it is where the crash happens. From here you can find your way backwards to the cause.

The array is `searchResults` and the index is given by `indexPath.row`. It would be great to get some insight into the row number but unfortunately there is no easy way to see the value of `indexPath.row` in the debugger.

You'll have to resort to using the debugger's command line interface, like a hacker whiz kid from the movies.

➤ Behind the **(lldb)** prompt, type **p indexPath.row** and press enter:



Printing the value of indexPath.row

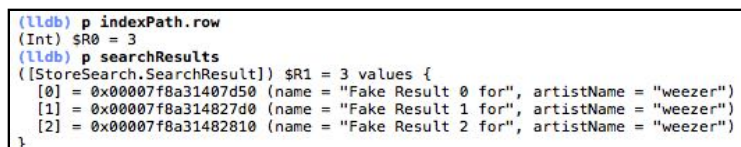
The output should be something like:

```
(Int) $R0 = 3
```

This means the value of `indexPath.row` is 3 and the type is `Int`. (You can ignore the `$R0` bit.)

Let's also find out how many items are in the array.

➤ Type **p searchResults** and press enter:



Printing the searchResults array

The output shows an array with three items.

You can now reason about the problem: the table view is asking for a cell for the fourth row (i.e. the one at index 3) but apparently there are only three rows in the data model (rows 0 through 2).

The table view knows how many rows there are from the value that is returned from `numberOfRowsInSection`, so maybe that method is returning the wrong number of rows. That is indeed the cause, of course, as you intentionally introduced the bug in that method.

I hope this illustrates how you should deal with crashes: first find out where the crash happens and what the actual error is, then reason your way backwards until you find the cause.

► Restore `numberOfRowsInSection` to what it was and then add a new outlet property to **SearchViewController.swift**:

```
@IBOutlet weak var searchBar2: UISearchBar!
```

► Open the storyboard and **Ctrl-drag** from Search View Controller to the Search Bar. Select **searchBar2** from the popup.

Now the search bar is also connected to this new `searchBar2` outlet. (It's perfectly fine for an object to be connected to more than one outlet at a time.)

► Remove the `searchBar2` outlet property from **SearchViewController.swift**.

This is a dirty trick on my part to make the app crash. The storyboard contains a connection to a property that no longer exists. (If you think this a convoluted example, then wait until you make this mistake in one of your own apps. It happens more often than you may think!)

► Run the app and it immediately crashes. The crash is "Thread 1: signal SIGABRT".

The Debug pane says:

```
*** Terminating app due to uncaught exception 'NSUnknownKeyException',
reason: '[<StoreSearch.SearchViewController 0x7ff47a6242c0>
setValue:forUndefinedKey:]: this class is not key value coding-compliant
for the key searchBar2.'
*** First throw call stack:
(
  0   CoreFoundation      0x00000001099a63f5 __exceptionPreprocess + 165
  1   libobjc.A.dylib      0x000000010b4d4bb7 objc_exception_throw + 45
  . . .
```

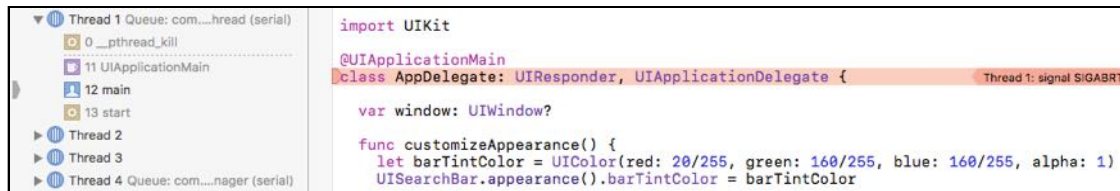
The first part of this message is very important: it tells you that the app was terminated because of an "NSUnknownKeyException". On some platforms exceptions are a commonly used error handling mechanism, but on iOS this is always a fatal error and the app is forced to halt.

The bit that should pique your interest is this:

this class is not key value coding-compliant for the key searchBar2

Hmm, that is a bit cryptic. It does mention searchBar2 but what does “key value-coding compliant” mean? I’ve seen this error enough times to know what is wrong but if you’re new to this game a message like that isn’t very enlightening.

So let’s see where Xcode thinks the crash happened:

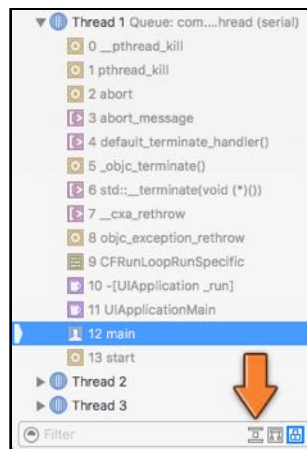


Crash in AppDelegate?

That also isn’t very useful. Xcode says the app crashed in **AppDelegate**, but that’s not really true.

Xcode goes through the **call stack** until it finds a method that it has source code for and that’s the one it shows. The call stack is the list of methods that have been called most recently. You can see it on the left of the Debugger window.

► Click the left-most icon at the bottom of the Debug navigator to see more info:



A more detailed call stack

The method at the top, `__pthread_kill`, was the last method that was called (it’s actually a function, not a method). It got called from `pthread_kill`, which was called from `abort`, which was called from `abort_message`, and so on, all the way back to the main function, which is the entry point of the app and the very first function that was called when the app started.

All of the methods and functions that are listed in this call stack are from system libraries, which is why they are grayed out. If you click on one, you'll get a bunch of unintelligible assembly code:



You cannot look inside the source code of system libraries

So clearly this approach is not getting you anywhere. However, there is another thing you can try and that is to set an **Exception Breakpoint**.

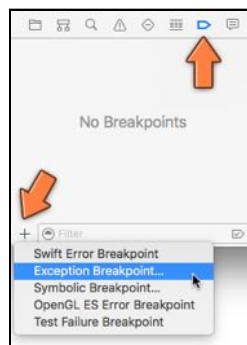
A **breakpoint** is a special marker in your code that will pause the app and jump into the debugger.

When your app hits a breakpoint, the app will pause at that exact spot. Then you can use the debugger to step line-by-line through your code in order to run it in slow motion. That can be a handy tool if you really cannot figure out why something crashes.

You're not going to step through code in this tutorial, but you can read more about it in the Xcode Overview guide, in the section Using the Debugger. You can find it in the iOS Developer Library at developer.apple.com.

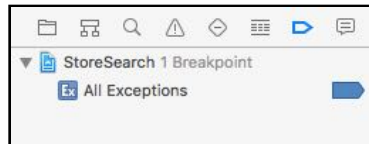
You are going to set a special breakpoint that is triggered whenever a fatal exception occurs. This will halt the program just as it is about to crash, which should give you more insight into what is going on.

➤ Switch to the **Breakpoint navigator** (the arrow-shaped button to the right of the Debug navigator) and click the **+** button at the bottom to add an Exception Breakpoint:



Adding an Exception Breakpoint

This will put a new breakpoint in the list:



After adding the Exception Breakpoint

► Now run the app again. It will still crash, but Xcode shows a lot more info:



Xcode now halts the app at the point the exception occurs

There are many more methods in the call stack now. Let's see if we can find some clues as to what is going on.

What catches my attention is the call to something called `[UIViewController _loadViewFromNibNamed:bundle:]`. That's a pretty good hint that this has error occurs when loading a nib file, or the storyboard in this case.

Using these hints and clues, and the somewhat cryptic error message that you got without the Exception Breakpoint, you can usually figure out what is making your app crash.

In this case we've established that the app crashes when it's loading the storyboard, and the error message mentioned "searchBar2". Put two and two together and you've got your answer.

A quick peek in the source code confirms that the `searchBar2` outlet no longer exists on the view controller but the storyboard still refers to it.

► Open the storyboard and in the **Connections inspector** disconnect Search View Controller from **searchBar2** to fix the crash. That's another bug squashed!

Note: Enabling the Exception Breakpoint means that you no longer get a useful error message in the Debug pane if the app crashes (because the breakpoint stops the app just before the exception happens). If sometime later during development your app crashes on another bug, you may want to

disable this breakpoint again to actually see the error message. You can do that from the Breakpoint navigator.

To summarize:

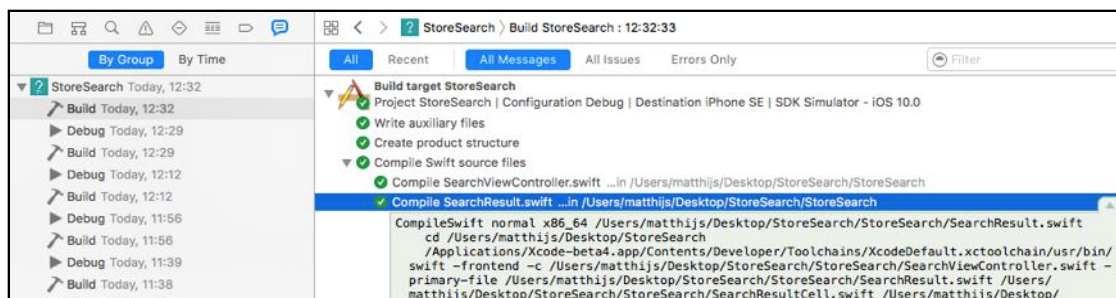
- If your app crashes with `EXC_BAD_INSTRUCTION` or `SIGABRT`, the Xcode debugger will often show you an error message and where in the code the crash happens.
- If Xcode thinks the crash happened on **AppDelegate** (not very useful!), enable the Exception Breakpoint to get more info.
- If the app crashes with a `SIGABRT` but there is no error message, then disable the Exception Breakpoint and make the app crash again. (Alternatively, click the **Continue program execution** button from the debugger toolbar a few times. That will also show the error message.)
- An `EXC_BAD_ACCESS` error usually means something went wrong with your memory management. An object may have been “released” one time too many or not “retained” enough. With Swift these problems are mostly a thing of the past because the compiler will usually make sure to do the right thing. However, it’s still possible to mess up if you’re talking to Objective-C code or low-level APIs.
- `EXC_BREAKPOINT` is not an error. The app has stopped on a breakpoint, the blue arrow pointing at the line where the app is paused. You set breakpoints to pause your app at specific places in the code, so you can examine the state of the app inside the debugger. The “Continue program execution” button resumes the app.

This should help you get to the bottom of most of your crashes!



The build log

If you’re wondering what Xcode actually does when it builds your app, then take a peek at the **Log navigator**. It’s the last icon in the navigator pane.



The Log navigator keeps track of your builds and debug sessions so you can look back at what happened. It even remembers the debug output of previous runs of the app.

Make sure **All Messages** is selected. To get more information about a particular log item, hover over it and click the little icon that appears on the right. The line will expand and you'll see exactly which commands Xcode executed and what the result was.

Should you run into some weird compilation problem, then this is the place for troubleshooting. Besides, it's interesting to see what Xcode is up to from time to time.



It's all about the networking

Now that the preliminaries are out of the way, you can finally get to the good stuff: adding networking to the app.

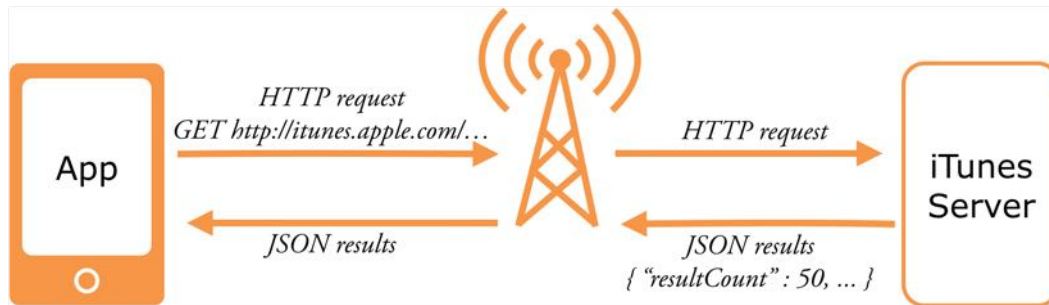
The iTunes store sells a lot of products: songs, e-books, movies, software, TV episodes... you name it. You can sign up as an affiliate and earn a commission on each sale that happens because you recommended a product (even your own apps!).

To make it easier for affiliates to find products, Apple made available a web service that queries the iTunes store. You're not going to sign up as an affiliate for this tutorial but you will use that free web service to perform searches.

So what is a **web service**? Your app (also known as the "client") will send a message over the network to the iTunes store (the "server") using the HTTP protocol.

Because the iPhone can be connected to different types of networks – Wi-Fi or a cellular network such as LTE, 3G, or GPRS – the app has to "speak" a variety of

networking protocols to communicate with other computers on the Internet.



The HTTP requests fly over the network

Fortunately you don't have to worry about any of that as the iPhone firmware will take care of this complicated subject matter. All you need to know is that you're using HTTP.

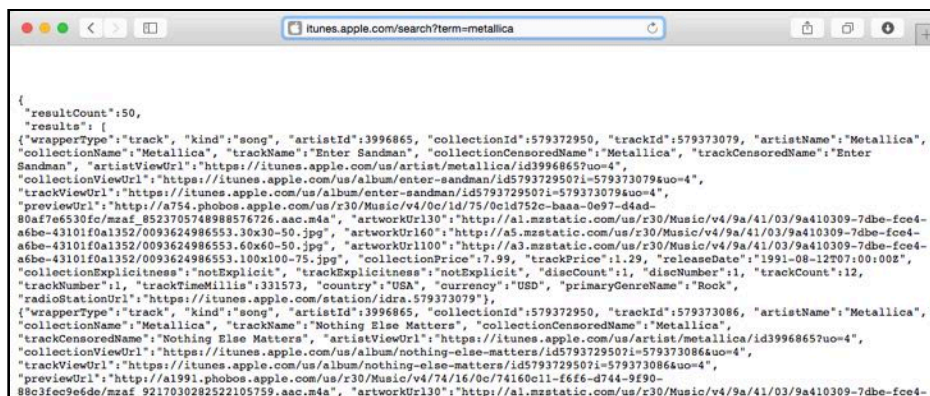
HTTP is the exact same protocol that your web browser uses when you visit a web site. In fact, you can play with the iTunes web service using a web browser. That's a great way to figure out how this web service works.

This trick won't work with all web services (some require "POST" requests instead of "GET" requests) but often you can get quite far with just a web browser.

Open your favorite web browser (I'm using Safari) and go to the following URL:

<http://itunes.apple.com/search?term=metallica>

The browser should show something like this:



Using the iTunes web service from the Safari web browser

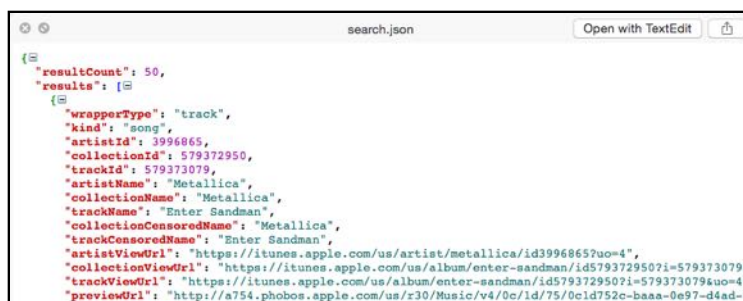
(It's also possible that the browser downloads these results into a file and puts it into your Downloads folder. If that happens, open it in a text editor or in Xcode.)

Those are the search results that the iTunes web service gives you. The data is in a format named JSON, which stands for JavaScript Object Notation.

JSON is commonly used to send structured data back-and-forth between servers and clients (i.e. apps). Another data format that you may have heard of is XML, but that's quickly going out of favor for JSON.

There are a variety of tools that you can use to make the JSON output more readable for mere humans. I have a Quick Look plug-in installed that renders JSON files in a colorful view (www.sagtau.com/quicklookjson.html).

You do need to save the output from the server to a **.json** file first and then open it from Finder by pressing the space bar:



A more readable version of the output from the web service

That makes a lot more sense.

Note: You can find extensions for Safari (and most other browsers) that can prettify JSON directly inside the browser. github.com/rfletcher/safari-json-formatter is a good one.

There are also dedicated tools on the Mac App Store, for example Visual JSON, that let you directly perform the request on the server and show the output in a structured and readable format.

A great online tool is codebeautify.org/jsonviewer.

Browse through the JSON text for a bit. You'll see that the server gave back a list of items, some of which are songs; others are audiobooks or music videos.

Each item has a bunch of data associated with it, such as an artist name ("Metallica", which is what you searched for), a track name, a genre, a price, a release date, and so on.

You'll store some of these fields in the `SearchResult` class so you can display them on the screen.

The results you get from the iTunes store might be different from mine. By default the search returns at most 50 items and since the store has quite a bit more than

fifty entries that match “metallica”, each time you do the search you may get back a different set of 50 results.

Also notice that some of these fields, such as `artistViewUrl` and `artworkUrl100` and `previewUrl` are links (URLs). For example, from the search result for the song “One” from the album “...And Justice for All”:

```
artistViewUrl:
https://itunes.apple.com/us/artist/metallica/id3996865?uo=4

artworkUrl100:
http://is1.mzstatic.com/image/thumb/Music6/v4/81/e1/fb/81e1fbe6-296c-5942-b410-7589de3af107/source/100x100bb.jpg

previewUrl:
http://a291.phobos.apple.com/us/r30/Music7/v4/f1/9c/78/f19c7823-85d0-48c7-8641-60fd9fcbf4bc/mzaf_3674189658242762920.plus.aac.p.m4a
```

Go ahead and copy-paste these URLs in your browser (use the ones from your own search results).

The `artistViewUrl` will open an iTunes Preview page for the artist, the `artworkUrl100` loads a thumbnail image, and the `previewUrl` opens a 30-second audio preview.

This is how the server tells you about additional resources. The images and so on are not embedded directly into the search results, but you’re given a URL that allows you to download them separately. Try some of the other URLs from the JSON data and see what they do!

Back to the original HTTP request. You made the web browser go to the following URL:

```
http://itunes.apple.com/search?term=the search term
```

You can add other parameters as well to make the search more specific. For example:

```
http://itunes.apple.com/search?term=metallica&entity=song
```

Now the results won’t contain any music videos or podcasts, only songs.

If the search term has a space in it you should replace it with a + sign, as in:

```
http://itunes.apple.com/search?term=angry+birds&entity=software
```

This searches for all apps that have something to do with angry birds (you may have heard of some of them).

The fields in the JSON results for this particular query are slightly different than before. There is no more `previewUrl` but there are several `screenshot` URLs per entry. Different kinds of products – songs, movies, software – return different types

of data.

That's all there is to it. You construct a URL to `itunes.apple.com` with the search parameters and then use that URL to make an HTTP request. The server will send JSON gobbledygook back to the app and you'll have to somehow turn that into `SearchResult` objects and put them in the table view. Let's get on it!



Synchronous networking = bad

Before you begin, I should point out that there is a bad way to do networking in your apps and a good way. The bad way is to perform the HTTP requests on your app's **main thread**.

This is simple to program but it will block the user interface and make your app unresponsive while the networking is taking place. Because it blocks the rest of the app, this is called synchronous networking.

Unfortunately, many programmers insist on doing networking the wrong way in their apps, which makes for apps that are slow and prone to crashing.

I will begin by demonstrating the easy-but-bad way, just to show you how *not* to do this. It's important that you realize the consequences of synchronous networking, so you will avoid it in your own apps.

After I have convinced you of the evilness of this approach, I will show you how to do it the right way. That only requires a small modification to the code but may require a big change in how you think about these problems.

Asynchronous networking (the right kind, with an "a") makes your apps much more responsive, but also brings with it additional complexity that you need to deal with.



Sending the HTTP request to the iTunes server

The to-do list for this section:

- Create the URL with the search parameters.
- Do the request on the iTunes server and see if you get any data back.
- Turn the JSON data into something more useful, i.e. `SearchResult` objects.

- Show these `SearchResult` objects in the table view.
- Take care of errors. There may be no or a very bad network connection, or the iTunes server may send back data that the app does not know how to interpret. The app should be able to recover from such situations.

You will not worry about downloading the artwork images for now; just the list of products will be plenty for our poor brains to handle.

➤ Add a new method to **`SearchViewController.swift`**:

```
func iTunesURL(searchText: String) -> URL {
    let urlString = String(format:
        "https://itunes.apple.com/search?term=%@", searchText)
    let url = URL(string: urlString)
    return url!
}
```

This first builds the URL as a string by placing the text from the search bar behind the “term=” parameter, and then turns this string into a URL object.

Because `URL(string)` is one of those failable initializers, it returns an optional. You force unwrap that using `url!` to return an actual URL object.

HTTPS vs. HTTP

Previously you used `http://` but here you’re using `https://`. The difference is that HTTPS is the secure, encrypted version of HTTP. It protects your users from eavesdropping. The underlying protocol is the same but any bytes that you’re sending or receiving are encrypted before they go out on the network.

As of iOS 9, apps should always use HTTPS. In fact, even if you specify an unprotected `http://` URL, iOS will still try to connect using HTTPS. If the server isn’t configured to speak HTTPS but only unprotected HTTP, then the network connection will fail.

You can ask to be exempt from this behavior in your Info.plist file, but that is generally not recommended. Later in the tutorial you’ll learn how to do this because the artwork images are hosted on a server that does not talk HTTPS.

➤ Change `searchBarSearchButtonClicked()` to:

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
    if !searchBar.text!.isEmpty {
        searchBar.resignFirstResponder()

        hasSearched = true
        searchResults = []

        let url = iTunesURL(searchText: searchBar.text!)
        print("URL: '\(url)')
    }
}
```

```

        tableView.reloadData()
    }
}

```

You've removed the code that creates the fake `SearchResult` items, and instead call the new `iTunesURL(searchText)` method. For testing purposes you log the `URL` object that this method returns.

This logic sits inside the if-statement so that none of this happens unless the user actually typed text into the search bar – it doesn't make much sense to search the iTunes store for "nothing".

Note: Don't get confused by all the exclamation points in the line,
`if !searchBar.text!.isEmpty`

The first one is the "logical not" operator because you want to go inside the if-statement if the text is not empty. The second exclamation point is for force unwrapping the value of `searchBar.text`, which is an optional. (It will never actually be `nil`, so it being an optional is a bit silly, but whaddya gonna do?)

► Run the app and type in some search text, for example "metallica" (or one of your other favorite metal bands), and press the Search button.

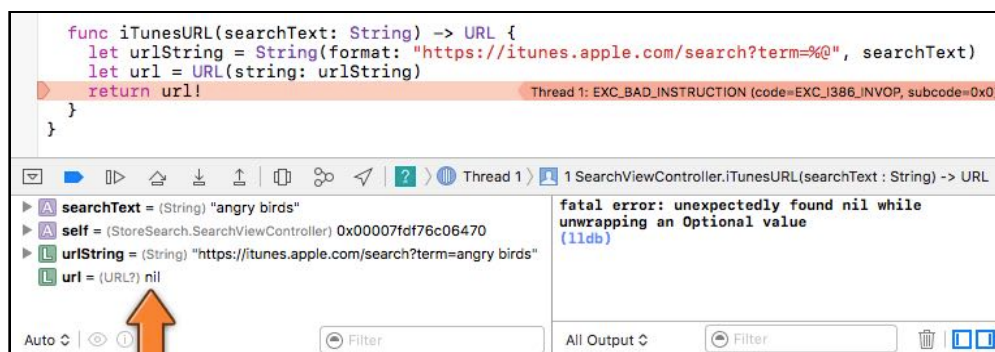
Xcode should now show this in its Debug pane:

```
URL: 'https://itunes.apple.com/search?term=metallica'
```

That looks good.

► Now type "angry birds" into the search box.

Whoops, the app crashes!



The crash after searching for "angry birds"

Look into the left-hand pane of the Xcode debugger and you'll see that the value of the `url` constant is `nil` (this may also show up as `0x0000...` followed by a whole

bunch of zeros).

The app apparently did not create a valid URL object. But why?

A space is not a valid character in a URL. Many other characters aren't valid either (such as the < or > signs) and therefore must be **escaped**. Another term for this is **URL encoding**.

A space, for example, can be encoded as the + sign (you did that earlier when you typed the URL into the web browser) or as the character sequence %20.

► Fortunately, String can do this encoding already, so you only have to add one extra statement to the app to make this work:

```
func iTunesURL(searchText: String) -> URL {
    let escapedSearchText = searchText.addingPercentEncoding(
        withAllowedCharacters: CharacterSet.urlQueryAllowed)!
    let urlString = String(format:
        "https://itunes.apple.com/search?term=%@", escapedSearchText)
    let url = URL(string: urlString)
    return url!
}
```

This calls the `addingPercentEncoding(withAllowedCharacters)` method to escape the special characters, which returns a new string that you use for the search term.

UTF-8 string encoding

This new string treats the special characters as being "UTF-8 encoded". It's important to know what that means because you'll run into this UTF-8 thing every once in a while when dealing with text.

There are many different ways to encode text. You've probably heard of ASCII and Unicode, the two most common encodings.

UTF-8 is a version of Unicode that is very efficient for storing regular text, but less so for special symbols or non-Western alphabets. Still, it's the most popular way to deal with Unicode text today.

Normally you don't have to worry about how your strings are encoded but when sending requests to a web service you need to transmit the text in the proper encoding. Tip: When in doubt, use UTF-8, it will almost always work.

► Run the app and search for "angry birds" again. This time a valid URL object can be created, and it looks like this:

```
URL: 'https://itunes.apple.com/search?term=angry%20birds'
```

The space has been turned into the character sequence %20. The % indicates an escaped character and 20 is the UTF-8 value for a space. Also try searching for terms with other special characters, such as # and * or even Emoji, and see what

happens.

Now that you have a URL object, you can do some actual networking!

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

```
func performStoreRequest(with url: URL) -> String? {
    do {
        return try String(contentsOf: url, encoding: .utf8)
    } catch {
        print("Download Error: \(error)")
        return nil
    }
}
```

The meat of this method is the call to `String(contentsOf, encoding)` that returns a new string object with the data it receives from the server at the other end of the URL.

Note that you're telling the app to interpret the data as UTF-8 text. Should the server send back the text in a different encoding then it will look like a garbled mess to your app. It's important that the sending and receiving sides agree on the encoding they are using!

Because things can go wrong – for example, the network may be down and the server cannot be reached – you're putting this in a `do-try-catch` block. If there is a problem, the code jumps to the catch section and the error variable contains more details about the error. You return `nil` to signal that the request failed.

➤ Add the following lines to `searchBarSearchButtonClicked()`, below the `print()` line:

```
if let jsonString = performStoreRequest(with: url) {
    print("Received JSON string '\(jsonString)'")
}
```

This invokes `performStoreRequest()` with the URL object as a parameter and returns the JSON data that is received from the server. If everything goes according to plan, this method returns a new string object that contains the JSON data that you're after. Let's try it out!

➤ Run the app and search for your favorite band. After a second or so, a whole bunch of data will be dumped to the Xcode Debug pane:

```
URL: 'http://itunes.apple.com/search?term=metallica'
Received JSON string '
{
  "resultCount":50,
  "results": [
    {"wrapperType":"track", "kind":"song", "artistId":3996865,
```

```
"collectionId":579372950, "trackId":579373079, "artistName":"Metallica",  
"collectionName":"Metallica", "trackName":"Enter Sandman",  
"collectionCensoredName":"Metallica", "trackCensoredName":"Enter  
Sandman",  
. . . and so on . . .
```

Congratulations! You've successfully made the app talk to a web service.

This prints the same stuff that you saw in the web browser earlier. Right now it's all contained in a single `String` object, which isn't really convenient for our purposes, but you'll convert it to a more useful format in a minute.

Of course, it's possible that you received an error. In that case, the output will be something like this:

```
Download Error: Error Domain=NSCocoaErrorDomain Code=256 "The operation  
couldn't be completed. (Cocoa error 256.)" UserInfo=0x7fc7e580bb10  
{NSURL=https://itunes.apple.com/search?term=metallica}
```

You'll add better error handling to the app later, but if you get such an error at this point, then make sure your computer is connected to the Internet (or your iPhone in case you're running the app on the device and not in the Simulator). Also try the URL directly in your web browser and see if that works.

Parsing JSON

Now that you have managed to download a chunk of JSON data from the server, what do you do with it?

JSON is a so-called *structured* data format. It typically consists of arrays and dictionaries that contain other arrays and dictionaries, as well as regular data such as string and numbers.

The JSON from the iTunes store roughly looks like this:

```
{  
  "resultCount": 50,  
  "results": [ . . . a bunch of other stuff . . . ]  
}
```

The `{ }` brackets surround a dictionary. This particular dictionary has two keys: `resultCount` and `results`. The first one, `resultCount`, has a numeric value. This is the number of items that matched your search query. By default the limit is a maximum of 50 items but as you shall later see you can increase this upper limit.

The `results` key contains an array, which is delineated by the `[]` brackets. Inside that array are more dictionaries, each of which describes a single product from the store. You can tell these things are dictionaries because they have the `{ }` brackets again.

Here are two of these items from the array:

```
{
  "wrapperType": "track",
  "kind": "song",
  "artistId": 3996865,
  "artistName": "Metallica",
  "trackName": "Enter Sandman",
  . . . and so on . . .
},
{
  "wrapperType": "track",
  "kind": "song",
  "artistId": 3996865,
  "artistName": "Metallica",
  "trackName": "Nothing Else Matters",
  . . . and so on . . .
},
```

Each product is represented by a dictionary with several keys. The values of the `kind` and `wrapperType` keys determine what sort of product this is: a song, a music video, an audiobook, and so on. The other keys describe the artist and the song itself.



The structure of the JSON data

To summarize, the JSON data represents a dictionary and inside that dictionary is an array of more dictionaries. Each of the dictionaries from the array represents one search result.

Currently all of this sits in a `String`, which isn't very handy, but using a so-called **JSON parser** you can turn this data into actual `Dictionary` and `Array` objects.



JSON or XML?

JSON is not the only structured data format out there. A slightly more formal standard is XML, which stands for Extensible Markup Language. Both formats serve the same purpose but they look a bit different. If the iTunes store would return its results as XML, the output would look more like this:

```
<?xml version="1.0" encoding="utf-8"?>
<iTunesSearch>
  <resultCount>5</resultCount>
  <results>
    <song>
      <artistName>Metallica</artistName>
      <trackName>Enter Sandman</trackName>
    </song>
    <song>
      <artistName>Metallica</artistName>
      <trackName>Nothing Else Matters</trackName>
    </song>
    . . . and so on . . .
  </results>
</iTunesSearch>
```

These days most developers prefer JSON because it's simpler than XML and easier to parse. But it's perfectly possible that if you want your app to talk to a particular web service you'll be expected to speak XML.



In the past, if you wanted to parse JSON it used to be necessary to include a third-party framework into your apps but these days iOS comes with its own JSON parser, so that's easy.

► Add the following method somewhere in **SearchViewController.swift**:

```
func parse(json: String) -> [String: Any]? {
    guard let data = json.data(using: .utf8, allowLossyConversion: false)
    else { return nil }

    do {
        return try JSONSerialization.jsonObject(
            with: data, options: []) as? [String: Any]
    } catch {
        print("JSON Error: \(error)")
        return nil
    }
}
```

You're using the `JSONSerialization` object here to convert the JSON search results to a Dictionary.

The dictionary is of type `[String: Any]`. The dictionary keys will always be strings but the values from these keys can be anything from a string to a number to a boolean. That's why the type of the values is `Any`.

Because the JSON data is currently in the form of a string, you have to put it into a `Data` object first. Again you have to be specific about what encoding to use: `UTF-8`.

The chance is small but it's possible that this conversion of string to `Data` fails – for example, if the text from the string cannot be represented in the encoding you've chosen. That is why you're using a guard statement.

`guard let` works like `if let`, it unwraps the optionals for you. But if unwrapping fails, i.e. if `json.data(...)` returns `nil`, the guard's `else` block is executed and you return `nil` to indicate that `parse(json)` failed. This "should" never happen in our app, but it's good to be vigilant about this kind of thing. (Never say never!)

If everything went OK – and 99.999% of the time it will! – you convert the `Data` object into a `Dictionary` using `JSONSerialization.jsonObject(...)`. Or at least, you *hope* you can convert it into a dictionary...



Assumptions cause trouble

When you write apps that talk to other computers on the Internet, one thing to keep in mind is that your conversational partners may not always say the things you expect them to say.

There could be an error on the server and instead of valid JSON data it may send back some error message. In that case, `JSONSerialization` will not be able to parse the data and the app will return `nil` from `parse(json)`.

Another thing that could happen is that the owner of the server changes the format of the data they send back. Usually this is done in a new version of the web service that runs on some other URL or that requires you to send along a "version" parameter. But not everyone is careful like that and by changing what the server does, they may break apps that depend on the data coming back in a specific format.

Just because `JSONSerialization` was able to turn the string into valid Swift objects, doesn't mean that it returns a `Dictionary`! It could have returned an `Array` or even a `String` or a number...

In the case of the iTunes store web service, the top-level object *should* be a `Dictionary`, but you can't control what happens on the server. If for some reason the server programmers decide to put `[]` brackets around the JSON data, then the top-level object will no longer be a `Dictionary` but an `Array`.

Being paranoid about these kinds of things and showing an error message in the unlikely event this happens is a lot better than your application suddenly crashing when something changes on a server that is outside of your control.

Just to be sure, you're using the `as?` cast to check that the object returned by `JSONSerialization` is truly a `Dictionary`. Should the conversion to a dictionary of `Strings` and `Any` objects fail, then the app doesn't burst into flames but simply returns `nil` to signal an error.

It's good to add checks like these to the app to make sure you get back what you expect. If you don't own the servers you're talking to, it's best to program defensively.



► Add the following lines to `searchBarSearchButtonClicked()`, inside the `if let jsonString` block:

```
if let jsonDictionary = parse(json: jsonString) {  
    print("Dictionary \(jsonDictionary)")  
}
```

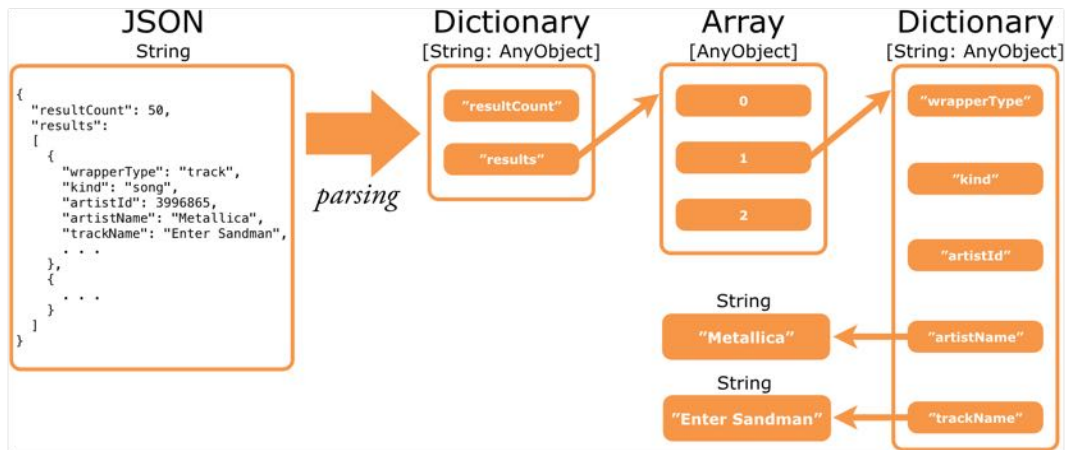
You simply call the new `parse(json)` method and print its return value.

► Run the app and search for something. The Xcode Debug pane now prints the following:

```
Dictionary [results: <__NSArrayI 0x...>(  
    {  
        artistId = 3996865;  
        artistName = Metallica;  
        kind = song;  
        trackName = "Enter Sandman";  
        . . . more fields . . .  
    },  
    {  
        artistId = 3996865;  
        artistName = Metallica;  
        kind = song;  
        trackName = "Nothing Else Matters";  
        . . . more fields . . .  
    },  
    . . . and so on . . .  
) , resultCount: 50]
```

This should look very familiar to the JSON data – which is not so strange because it represents the exact same thing – except that now you're looking at the contents of a Swift `Dictionary` object.

You have converted a bunch of text that was all packed together in a single string into actual objects that you can use.



Parsing JSON turns text into objects

Let's add an alert to handle potential errors. It's inevitable that something goes wrong somewhere, so it's best to be prepared.

➤ Add the following method:

```

func showNetworkError() {
    let alert = UIAlertController(
        title: "Whoops...",
        message:
            "There was an error reading from the iTunes Store. Please try again.",
        preferredStyle: .alert)

    let action = UIAlertAction(title: "OK", style: .default, handler: nil)
    alert.addAction(action)

    present(alert, animated: true, completion: nil)
}
  
```

Nothing you haven't seen before; it simply presents an alert controller with an error message.

➤ Change searchBarSearchButtonClicked() to the following:

```

func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
    if !searchBar.text!.isEmpty {
        ...

        if let jsonString = performStoreRequestWithURL(url) {
            if let jsonDictionary = parseJSON(jsonString) {
                print("Dictionary \(jsonDictionary)")

                tableView.reloadData() // this has changed
            }
        }
    }
}
  
```

```
        }  
        return  
    }  
    }  
    showNetworkError()           // this is new  
}
```

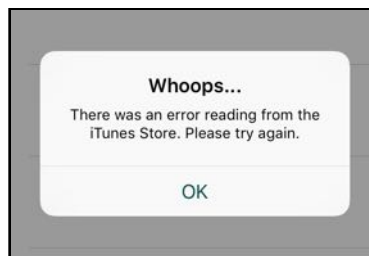
You moved the call to `tableView.reloadData()` into the innermost `if let` statement and added a `return` statement. If the code made it there, then everything went OK. But if something goes wrong, one of these `if let` statements is false, and you call `showNetworkError()` to show an alert box.

If you did everything correctly up to this point then the web service should always have worked. Still it's a good idea to test a few error situations, just to make sure the error handling is working for those unlucky users with bad network connections.

► Try this: In the `iTunesURL(searchText)` method, temporarily change the `itunes.apple.com` part of the URL to `"NOMOREitunes.apple.com"`.

You should now get an error alert when you try a search because no such server exists at that address. This simulates the iTunes server being down. Don't forget to change the URL back when you're done testing.

Tip: To simulate no network connection you can pull the network cable and/or disable Wi-Fi on your Mac, or run the app on your device in Airplane Mode.



The app shows an alert when there is a network error

Interestingly enough, a little while ago I was also able to make the app fail simply by searching for "photoshop". The Xcode Debug pane said:

```
JSON Error: Error Domain=NSCocoaErrorDomain Code=3840 "The operation  
couldn't be completed. (Cocoa error 3840.)" (Missing low code point in  
surrogate pair around character 92893.) UserInfo=0x6eb3110  
{NSDebugDescription=Missing low code point in surrogate pair around  
character 92893.}
```

This may sound like gibberish, but it means that `JSONSerialization` was unable to convert the data to Swift objects because it thinks there is some error in the data. However, when I typed the URL into my web browser it seemed to return valid JSON data, which I verified with JSONLint (jsonlint.com).

So who was right? It could have been a bug in JSONSerialization or it could be that the iTunes web service did something naughty... As of the current revision of this tutorial, searching for “photoshop” works again.

In any case, it should be obvious that when you’re doing networking things can – and will! – go wrong, often in unexpected ways.

Turning the JSON into SearchResult objects

So far you’ve managed to send a request to the iTunes web service and you parsed the JSON data into a bunch of Dictionary objects. That’s a great start, but now you are going to turn this into an array of SearchResult objects because they’re much easier to work with.

The iTunes store sells different kinds of products – songs, e-books, software, movies, and so on – and each of these has its own structure in the JSON data. A software product will have screenshots but a movie will have a video preview. The app will have to handle these different kinds of data.

You’re not going to support everything the iTunes store has to offer, only these items:

- Songs, music videos, movies, TV shows, podcasts
- Audio books
- Software (apps)
- E-books

The reason I have split them up like this is because that’s how the iTunes store does it. Songs and music videos, for example, share the same set of fields, but audiobooks and software have different data structures. The JSON data makes this distinction using the kind and wrapperType fields.

➤ Add a new method, `parse(dictionary)`, to **SearchViewController.swift**:

```
func parse(dictionary: [String: Any]) {  
    // 1  
    guard let array = dictionary["results"] as? [Any] else {  
        print("Expected 'results' array")  
        return  
    }  
    // 2  
    for resultDict in array {  
        // 3  
        if let resultDict = resultDict as? [String: Any] {  
            // 4  
            if let wrapperType = resultDict["wrapperType"] as? String,  
                let kind = resultDict["kind"] as? String {  
                print("wrapperType: \(wrapperType), kind: \(kind)")  
            }  
        }  
    }  
}
```

```
}  
}
```

This method goes through the top-level dictionary and looks at each search result in turn. Here's what happens step-by-step:

1. First there is a bit of defensive programming to make sure the dictionary has a key named `results` that contains an array. It probably will, but better safe than sorry.
2. Once it is satisfied that array exists, the method uses a `for in` loop to look at each of the array's elements in turn.
3. Each of the elements from the array is another dictionary. A small wrinkle: the type of `resultDict` isn't `Dictionary` as we'd like it to be, but `Any`, because the contents of the array could in theory be anything.

To make sure these objects really do represent dictionaries, you have to cast them to the right type first. You're using the optional cast `as?` here as another defensive measure. In theory it's possible `resultDict` doesn't actually hold a `[String: Any]` dictionary and then you don't want to continue.

4. For each of the dictionaries, you print out the value of its `wrapperType` and `kind` fields. Indexing a dictionary always gives you an optional, which is why you're using `if let` to unwrap these two values. And because the dictionary only contains values of type `Any`, you also cast to the more useful `String`.

► Call this method from `searchBarSearchButtonClicked()`, just before the line that reloads the table view.

```
parse(dictionary: jsonDictionary)
```

► Run the app and do a search. Look at the Xcode output.

When I did this, Xcode showed three different types of products, with the majority of the results being songs. What you see may vary, depending on what you search for.

```
wrapperType: track, kind: song  
wrapperType: track, kind: feature-movie  
wrapperType: track, kind: music-video  
...
```

To turn these things into `SearchResult` objects, you're going to look at value of the `wrapperType` field first. If that is "track" then you know that the product in question is a song, movie, music video, podcast or episode of a TV show.

Other values for `wrapperType` are "audiobook" for audio books and "software" for apps, and you will interpret these differently than "tracks".

But before you get to that, let's first add some new properties to the `SearchResult`

object.



Always check the documentation

If you were wondering how I knew how to interpret the data from the iTunes web service, or even how to make the URLs to use the service in the first place, then you should realize there is no way you can be expected to use a web service if there is no documentation.

Fortunately, for the iTunes store web service there is a pretty good document that explains how to use it:

<https://affiliate.itunes.apple.com/resources/documentation/itunes-store-web-service-search-api/>

Just reading the docs is often not enough. You have to play with the web service for a bit to know what you can and cannot do.

There are some things that the StoreSearch app needs to do with the search results that were not clear from reading the documentation. For example, e-books do not include a `wrapperType` field for some reason.

So first read the docs and then play with it. That goes for any API, really, whether it's something from the iOS SDK or a web service.



A better SearchResult

The current `SearchResult` class only has two properties: `name` and `artistName`. As you've seen, the iTunes store returns a lot more information than that, so you'll need to add a few new properties.

➤ Add the following instance variables to **SearchResult.swift**:

```
var artworkSmallURL = ""
var artworkLargeURL = ""
var storeURL = ""
var kind = ""
var currency = ""
var price = 0.0
var genre = ""
```

You're not including *everything* that the iTunes store returns, only the fields that are interesting to this app.

SearchResult stores two artwork URLs, one for a 60×60 pixel image and the other for a 100×100 pixel image. It also stores the kind and genre of the item, its price and the currency (US dollar, Euro, British Pounds, etc.), as well as a link to the product's page on the iTunes store itself.

All right, now that you have some place to put this data, let's get it out of the dictionaries and into the SearchResult objects.

► Back in **SearchViewController.swift**, make the following changes to the `parse(dictionary)` method:

```
func parse(dictionary: [String: Any]) -> [SearchResult] {
    guard let array = dictionary["results"] as? [Any] else {
        print("Expected 'results' array")
        return []
    }

    var searchResults: [SearchResult] = []

    for resultDict in array {
        . . .
    }

    return searchResults
}
```

You're making the method return an array of SearchResult objects. (If something went wrong during parsing, it simply returns an empty array.)

► Still in `parse(dictionary)`, change the inside of the `if let resultDict` statement to the following. The existing code marked with `//4` should be replaced by:

```
var searchResult: SearchResult?

if let wrapperType = resultDict["wrapperType"] as? String {
    switch wrapperType {
        case "track":
            searchResult = parse(track: resultDict)
        default:
            break
    }
}

if let result = searchResult {
    searchResults.append(result)
}
```

If the found item is a "track" then you create a SearchResult object for it, using a new method `parse(track)`, and add it to the `searchResults` array.

For any other types of products, the temporary variable `searchResult` remains `nil`

and doesn't get added to the array (that's why it's an optional).

You'll be adding more wrapper types to the switch soon but for now you're limiting it to just the "track" type, which is used for songs, movies, and TV episodes.

➤ Also add the `parse(track)` method:

```
func parse(track dictionary: [String: Any]) -> SearchResult {
    let searchResult = SearchResult()

    searchResult.name = dictionary["trackName"] as! String
    searchResult.artistName = dictionary["artistName"] as! String
    searchResult.artworkSmallURL = dictionary["artworkUrl60"] as! String
    searchResult.artworkLargeURL = dictionary["artworkUrl100"] as! String
    searchResult.storeURL = dictionary["trackViewUrl"] as! String
    searchResult.kind = dictionary["kind"] as! String
    searchResult.currency = dictionary["currency"] as! String

    if let price = dictionary["trackPrice"] as? Double {
        searchResult.price = price
    }
    if let genre = dictionary["primaryGenreName"] as? String {
        searchResult.genre = genre
    }
    return searchResult
}
```

It's a big chunk of code but what happens here is quite simple. You first instantiate a new `SearchResult` object, then get the values out of the dictionary and put them into the `SearchResult`'s properties.

All of these things are strings except the track price, which is a number. Because the dictionary is defined as having `Any` values, you first need to cast to `String` and `Double` here.

Note: Something else interesting is going on here, did you spot it? You've learned that indexing a dictionary always gives you an optional. If that is true, then why don't you need to use `if let` with lines such as these:

```
searchResult.name = dictionary["trackName"] as! String
```

After all, `dictionary["trackName"]` is an optional but `searchResult.name` is definitely not... How come you can assign an optional value to a non-optional?

The trick is in the cast. When you write `"as! <something>"`, you're telling the compiler that you're sure this isn't ever going to be `nil`. (Of course if it turns out you're wrong, the app will crash.)

If you wanted to keep the optional status, you'd have to write `"as! String?"` or `"as? String"`. The former means you're casting to an optional `String`; the latter means you're trying to cast to a regular `String` but it might fail and be `nil` because it's not really a string. It's a subtle difference.

The reason you're using `if let` for the `trackPrice` and `primaryGenreName` is that sometimes these fields are missing from the JSON data.

With these latest changes, `parse(dictionary)` returns an array of `SearchResult` objects, but you're not doing anything with that array yet.

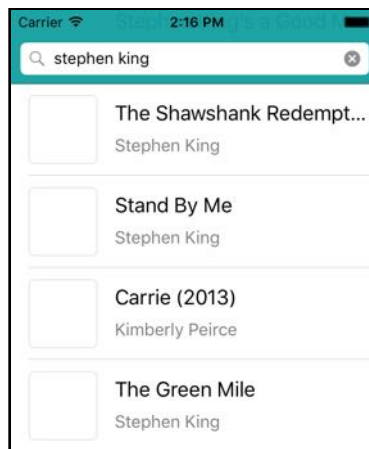
► In `searchBarSearchButtonClicked()`, change the line that calls `parse(dictionary)` to the following:

```
searchResults = parse(dictionary: jsonDictionary)
```

Now the returned array is placed into the instance variable and the table view can show the actual search result objects.

► Run the app and search for your favorite musician. After a second or so you should see a whole bunch of results appear in the table. Cool!

You don't have to search for music, of course. You can also search for names of books, software, or authors. For example, a search for Stephen King brings up results such as these:



The results from the search now show up in the table

The search results may include podcasts, songs, or other related products. It would be useful to make the table view display what type of product it is showing, so let's improve `tableView(cellForRowAt)` a little.

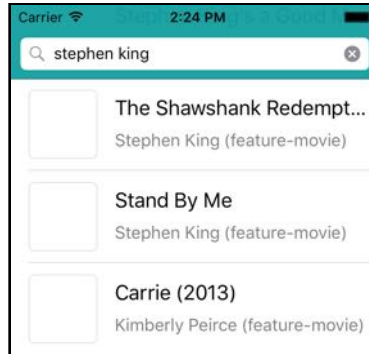
► In `tableView(cellForRowAt)`, change the line that sets `cell.artistNameLabel` to the following:

```
if searchResult.artistName.isEmpty {  
    cell.artistNameLabel.text = "Unknown"  
} else {  
    cell.artistNameLabel.text = String(format: "%@ (%@)",  
                                       searchResult.artistName, searchResult.kind)  
}
```

The first change is that you now check that the `SearchResult`'s `artistName` is not empty. When testing the app I noticed that sometimes a search result did not

include an artist name. In that case you make the cell say "Unknown".

You also add the value of the `kind` property to the artist name label, which should tell the user what kind of product they're looking at:



They're not books...

There is one problem with this. The value of `kind` comes straight from the server and it is more of an internal name than something you'd want to show directly to the user.

What if you want it to say "Movie" instead, or maybe you want to translate the app to another language (something you'll do later in this tutorial). It's better to convert this internal identifier ("feature-movie") into the text that you want to show to the user ("Movie").

➤ Add this new method:

```
func kindForDisplay(_ kind: String) -> String {
    switch kind {
    case "album": return "Album"
    case "audiobook": return "Audio Book"
    case "book": return "Book"
    case "ebook": return "E-Book"
    case "feature-movie": return "Movie"
    case "music-video": return "Music Video"
    case "podcast": return "Podcast"
    case "software": return "App"
    case "song": return "Song"
    case "tv-episode": return "TV Episode"
    default: return kind
    }
}
```

These are the types of products that this app understands.

It's possible that I missed one or that the iTunes Store adds a new product type at some point. If that happens, the switch jumps to the `default:` case and you'll simply return the original `kind` value (and hopefully fix this in an update of the app).

- In `tableView(cellForRowAt)`, change the line that sets the artist name label to:

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

Now the text inside the parentheses is no longer the internal identifier from the iTunes web service, but the one you gave it:



The product type is a bit more human-friendly

All right, let's put in the other types of products. This is very similar to what you just did.

- Add the following methods below `parse(track)`. Feel free to copy-paste the code from `parse(track)` three times, but be careful about the differences between these methods!

```
func parse(audiobook dictionary: [String: Any]) -> SearchResult {
    let searchResult = SearchResult()
    searchResult.name = dictionary["collectionName"] as! String
    searchResult.artistName = dictionary["artistName"] as! String
    searchResult.artworkSmallURL = dictionary["artworkUrl60"] as! String
    searchResult.artworkLargeURL = dictionary["artworkUrl100"] as! String
    searchResult.storeURL = dictionary["collectionViewUrl"] as! String
    searchResult.kind = "audiobook"
    searchResult.currency = dictionary["currency"] as! String

    if let price = dictionary["collectionPrice"] as? Double {
        searchResult.price = price
    }
    if let genre = dictionary["primaryGenreName"] as? String {
        searchResult.genre = genre
    }
    return searchResult
}
```

```
func parse(software dictionary: [String: Any]) -> SearchResult {
    let searchResult = SearchResult()
    searchResult.name = dictionary["trackName"] as! String
    searchResult.artistName = dictionary["artistName"] as! String
    searchResult.artworkSmallURL = dictionary["artworkUrl60"] as! String
    searchResult.artworkLargeURL = dictionary["artworkUrl100"] as! String
    searchResult.storeURL = dictionary["trackViewUrl"] as! String
    searchResult.kind = dictionary["kind"] as! String
    searchResult.currency = dictionary["currency"] as! String

    if let price = dictionary["price"] as? Double {
```



```

        searchResult.price = price
    }
    if let genre = dictionary["primaryGenreName"] as? String {
        searchResult.genre = genre
    }
    return searchResult
}

```

```

func parse(ebook dictionary: [String: Any]) -> SearchResult {
    let searchResult = SearchResult()
    searchResult.name = dictionary["trackName"] as! String
    searchResult.artistName = dictionary["artistName"] as! String
    searchResult.artworkSmallURL = dictionary["artworkUrl60"] as! String
    searchResult.artworkLargeURL = dictionary["artworkUrl100"] as! String
    searchResult.storeURL = dictionary["trackViewUrl"] as! String
    searchResult.kind = dictionary["kind"] as! String
    searchResult.currency = dictionary["currency"] as! String

    if let price = dictionary["price"] as? Double {
        searchResult.price = price
    }
    if let genres: Any = dictionary["genres"] {
        searchResult.genre = (genres as! [String]).joined(separator: ", ")
    }
    return searchResult
}

```

Two interesting points here:

- Audio books don't have a "kind" field, so you have to set the kind property to "audiobook" yourself.
- E-books don't have a "primaryGenreName" field, but an array of genres. You use the `joined(separator)` method to glue these genre names into a single string, separated by commas.

You still need to call these new methods, based on the value of the `wrapperType` field.

➤ Change the `if let wrapperType` statement in `parse(dictionary)` to:

```

if let wrapperType = resultDict["wrapperType"] as? String {
    switch wrapperType {
    case "track":
        searchResult = parse(track: resultDict)
    case "audiobook":
        searchResult = parse(audiobook: resultDict)
    case "software":
        searchResult = parse(software: resultDict)
    default:
        break
    }
} else if let kind = resultDict["kind"] as? String, kind == "ebook" {
    searchResult = parse(ebook: resultDict)
}

```

For some reason, e-books do not have a `wrapperType` field, so in order to determine whether something is an e-book you have to look at the `kind` field instead.

Depending on the value of `wrapperType` or `kind`, you call one of the parse methods to get a `SearchResult` object.

If there is a `wrapperType` or `kind` that the app does not support, no `SearchResult` object gets created, the value of `searchResult` is `nil`, and you simply skip that item.

Default and break

Switch statements often have a `default:` case at the end that just says `break`.

In Swift, a `switch` must be exhaustive, meaning that it must have a case for all possible values of the thing that you're looking at.

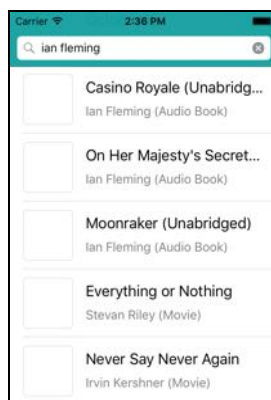
Here you're looking at `wrapperType`. Swift needs to know what to do when `wrapperType` is not "track", "audiobook", or "software". That's why you're required to include the `default:` case, as a catchall for any other possible values of `wrapperType`.

Because a case cannot be empty in Swift, you add a `break` statement to keep the compiler happy. The `break` doesn't do anything – it just says "Nothing to see here, move along."

By the way: unlike in other languages, the case statements in Swift do not need to say `break` at the end. They do not automatically "fall through" from one case to the other as they do in Objective-C.

► Run the app and search for software, audio books or e-books to see that the parsing code works. It can take a few tries before you find some because of the enormous quantity of products on the store.

Later in this tutorial you'll add a control that lets you pick the type of products that you want to search for, which makes it a bit easier to find just e-books or audiobooks.



The app shows a varied range of products now

Sorting the search results

It would be nice to sort the search results alphabetically. That's quite easy, actually. Array already has a method to sort itself – all you have to do is tell it what to sort on.

► In `searchBarSearchButtonClicked()`, between the lines that call `parse(dictionary)` and reload the table view, add the following:

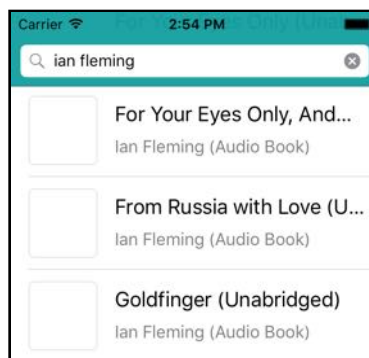
```
searchResults.sort(by: { result1, result2 in
    return result1.name.localizedStandardCompare(
        result2.name) == .orderedAscending
})
```

Before reloading the table, you first call `sort(by)` on the `searchResults` array with a closure that determines the sorting rules (the code in between the `{ }` brackets). This is identical to what you did in the Checklists tutorial to sort the to-do lists.

In order to sort the contents of the `searchResults` array, the closure will compare the `SearchResult` objects with each other and return `true` if `result1` comes before `result2`. The closure is called repeatedly on different pairs of `SearchResult` objects until the array is completely sorted.

The actual sorting rule calls `localizedStandardCompare()` to compare the names of the `SearchResult` objects. Because you used `.orderedAscending`, the closure returns `true` only if `result1.name` comes before `result2.name` – in other words, the array gets sorted from A to Z.

► Run the app and verify that the search results are sorted alphabetically.



The search results are sorted by name

Sorting was pretty easy to add but there is an even easier way to write this.

► Change the sorting code to:

```
searchResults.sort { $0.name.localizedStandardCompare($1.name)
    == .orderedAscending }
```

This uses the *trailing* closure syntax to put the closure behind the method name, rather than inside the traditional () parentheses as a parameter. It's a small improvement in readability.

More importantly, inside the closure you're no longer referring to the two `SearchResult` objects by name but with the special syntax `$0` and `$1`. Using these shorthand symbols instead of full parameter names is common in Swift closures. There is also no longer a return statement.

➤ Verify that this works.

Believe it or not, you can do even better. Swift has a very cool feature called **operator overloading**. It allows you to take the standard operators such as `+` and `*` and apply them to your own objects. You can even create completely new operator symbols.

It's not a good idea to go overboard with this feature and make operators do something completely unexpected – don't overload `/` to do multiplications, eh? – but it comes in very handy when doing sorting.

➤ Open **`SearchResult.swift`** and add the following code, outside of the class:

```
func < (lhs: SearchResult, rhs: SearchResult) -> Bool {  
    return lhs.name.localizedStandardCompare(rhs.name) == .orderedAscending  
}
```

This should look familiar! You're creating a function named `<` that contains the same code as the closure from earlier. This time the two `SearchResult` objects are called `lhs` and `rhs`, for left-hand side and right-hand side, respectively.

You have now overloaded the less-than operator so that it takes two `SearchResult` objects and returns `true` if the first one should come before the second, and `false` otherwise. Like so:

```
searchResultA.name = "Waltz for Debby"  
searchResultB.name = "Autumn Leaves"  
  
searchResultA < searchResultB // false  
searchResultB < searchResultA // true
```

➤ Back in **`SearchViewController.swift`**, change the sorting code to:

```
searchResults.sort { $0 < $1 }
```

That's pretty sweet. Using the `<` operator makes it very clear that you're sorting the items from the array in ascending order. But wait, you can write it even shorter:

```
searchResults.sort(by: <)
```

Wow, it doesn't get much simpler than that! This line literally says, "Sort this array in ascending order". Of course, this only works because you added your own `func <`

to overload the less-than operator so it takes two `SearchResult` objects and compares them.

► Run the app again and make sure everything is still sorted.

Exercise. See if you can make the app sort by the artist name instead. ■

Exercise. Try to sort in descending order, from Z to A. Tip: use the `>` operator. ■

Excellent! You made the app talk to a web service and you were able to convert the data that was received into your own data model objects.

The app may not support every product that's shown on the iTunes store, but I hope it illustrates the principle of how you can take data that comes in slightly different forms and convert it to objects that are more convenient to use in your own apps.

Feel free to dig through the web service API documentation to add the remaining items that the iTunes store sells: <https://affiliate.itunes.apple.com/resources/documentation/itunes-store-web-service-search-api/>

► Commit your changes.

You can find the project files for this section under **03 - Using Web Service** in the tutorial's Source Code folder.



SDKs for APIs

Often third-party services already have their own SDK (Software Development Kit) that lets you talk to their web service. In that case you don't have to write your own networking and JSON parsing code but you simply add a framework to your app and use the classes from that framework.

Some examples:

- Facebook <https://developers.facebook.com/docs/ios>
- Wordnik <http://developer.wordnik.com>
- Amazon Web Services <https://aws.amazon.com/mobile/sdk/>

and many others.

If you're really lucky, support for the web service is already built into iOS itself, such as the Social Framework that makes it very easy to put Twitter and Facebook into your apps.



Asynchronous networking

That wasn't so bad, was it? Yes it was, and I'll show you why! Did you notice that whenever you performed a search, the app became unresponsive?

While the network request was taking place, you could not scroll the table view up or down, or type anything new into the search bar. The app was completely frozen for a few seconds.

You may not have seen this if your network connection was very fast but if you're using your iPhone out in the wild, the network will be a lot slower than your home or office Wi-Fi, and a search can easily take ten seconds or more.

So what if the app is unresponsive while the search is taking place? After all, there is nothing for the user to do at that point anyway...

True, but to most users an app that does not respond is an app that has crashed. The screen looks empty, there is no indication of what is going on, and even an innocuous gesture such as sliding your finger up and down does not bounce the table view like you'd expect it to.

Conclusion: the app has crashed. The user will press the home button and try again – or more likely, delete your app, give it a bad rating on the App Store, and switch to a competing app.

Still not convinced? Let's slow down the network connection to pretend the app is running on an iPhone that someone may be using on a bus or in a train, not in the ideal conditions of a fast home or office network.

First off, you'll increase the amount of data that the app will get back. By adding a "limit" parameter to the URL you set the maximum number of results that the web service will return. The default value is 50, the maximum is 200.

► In `itunesURL(searchText)`, change the following line:

```
let urlString = String(format:
    "https://itunes.apple.com/search?term=%@&limit=200", escapedSearchText)
```

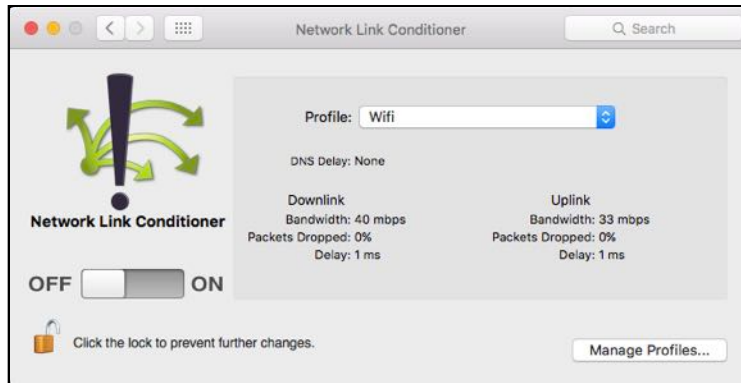
You added `&limit=200` to the URL. Just so you know, parameters in URLs are separated by the `&` sign, also known as the "and" sign.

► If you run the app now, the search should be quite a bit slower.

Still too fast? Then use the **Network Link Conditioner**. This is a pane in your System Preferences window that lets you simulate different network conditions,

including bad cell phone networks.

➤ Open the **System Preferences** on your Mac and locate **Network Link Conditioner** (it should be at the bottom).



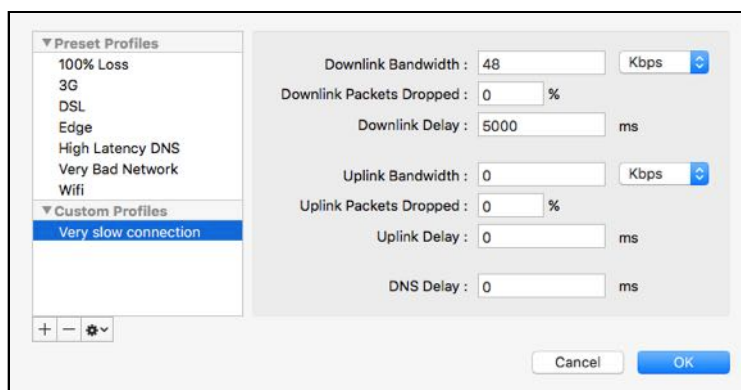
The Network Link Conditioner preference pane

If you don't have it installed yet, follow these instructions: From the **Xcode** menu choose **Open Developer Tool → More Developer Tools...** This opens the Apple developer website (you may need to login first). From the Downloads for Apple Developers page, download the latest **Additional Tools for Xcode 8** package. Open the downloaded file and under **Hardware** double-click **Network Link Conditioner.prefPane** to install it.

Let's simulate a really slow connection.

➤ Click on **Manage Profiles** and create a new profile with the following settings:

- Name: **Very slow connection**
- Downlink Bandwidth: **48 Kbps**
- Downlink Packets Dropped: **0 %**
- Downlink Delay: **5000 ms** (i.e. 5 seconds)



Adding the profile for a very slow connection

Press **OK** to add this profile and return to the main page. Make sure this new profile is selected and flick the switch to ON to start.

► Now run the app and search for something. The Network Link Conditioner tool will delay the HTTP request by 5 seconds in order to simulate a slow connection, and then downloads the data at a very slow speed.

Tip: If the download still appears very fast, then try searching for some term you haven't used before; the system may be caching the results from a previous search.

Notice how the app totally doesn't respond during this time? It feels like something is wrong. Did the app crash or is it still doing something? It's impossible to tell and very confusing to your users when this happens.

Even worse, if your program is unresponsive for too long, iOS may actually kill it by force, in which case it really did crash. You don't want that to happen!

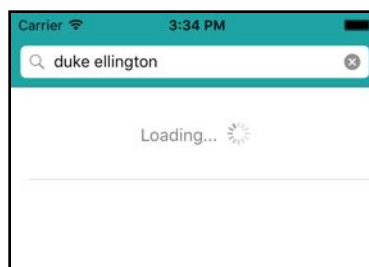
"Ah," you say, "let's show some type of animation to let the user know that the app is communicating with a server. Then at least they will know that the app is busy."

That sounds like a decent thing to do, so let's get to it.

Tip: Even better than pretending to have a lousy connection on the Simulator is to use Network Link Conditioner on your device, so you can also test bad network connections on your actual iPhone. You can find it under **Settings** → **Developer** → **Network Link Conditioner**. Using these tools to test whether your app can deal with real-world network conditions is a must! Not every user has the luxury of broadband...

The activity indicator

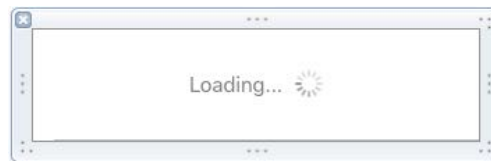
You've used the spinning activity indicator before in MyLocations to show the user that the app was busy. Let's create a new table view cell that you'll show while the app is querying the iTunes store. It will look like this:



The app shows that it is busy

- Create a new, empty nib file. Call it **LoadingCell.xib**.
- Drag a new **Table View Cell** into the canvas. Set its width to 320 points and its height to 80 points.
- Set the reuse identifier of the cell to **LoadingCell** and set the **Selection** attribute to **None**.
- Drag a new **Label** into the cell. Rename it to **Loading...** and change the font to **System 15**. The label's text color should be 50% opaque black.
- Drag a new **Activity Indicator View** into the cell and put it next to the label. Set its **Style** to **Gray** and give it the **Tag** 100.

The design looks like this:



The design of the LoadingCell nib

To make this cell work properly on the larger iPhone 6 and 7 models you'll add constraints that keep the label and the activity spinner centered in the cell. The easiest way to do this is to place these two items inside a container view and center that.

- Select both the Label and the Activity Indicator View (hold down ⌘ to make a multiple selection). From the Xcode menu bar, choose **Editor** → **Embed In** → **View**. This puts a larger, white, view behind them.



The label and the spinner now sit in a container view

- With this container view selected, click the **Align** button and put checkmarks in front of **Horizontally in Container** and **Vertically in Container** to make new constraints.



The container view has red constraints

You end up with a number of red constraints. That's no good; we want to see blue ones. The reason your new constraints are red is that Auto Layout does not know yet how large this container view should be; you've only added constraints for the view's position, not its size.

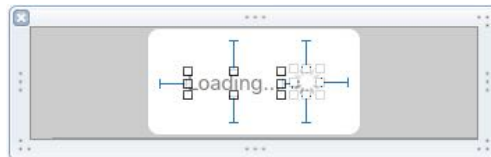
To fix this, you're going to add constraints to the label and activity indicator as well, so that the width and height of the container view are determined by the size of the two things inside it.

That is especially important for later when you're going to translate the app to another language. If the Loading... text becomes larger or smaller, then so should the container view, in order to stay centered inside the cell.

➤ Select the label and click the **Pin** button. Simply pin it to all four sides. Leave Update Frames set to **None** for now and press **Add 4 Constraints**.

➤ Repeat this for the Activity Indicator View. You don't need to pin it to the left because that constraint already exists (pinning the label added it).

Now the constraints for the label and the activity indicator should be all blue.



The label and spinner have blue constraints

At this point, the container view may still have orange lines. If so, select it and choose **Editor** → **Resolve Auto Layout Issues** → **Update Frames** (under Selected Views). This will move the container view into the position dictated by its constraints.

Cool, you now have a cell that automatically adjusts itself to any size device.

To make this special table view cell appear you'll follow the same steps as for the "Nothing Found" cell.

➤ Add the following line to the struct `TableViewCellIdentifiers` in **SearchViewController.swift**:

```
static let loadingCell = "LoadingCell"
```

- And register the nib in viewDidLoad():

```
cellNib = UINib(nibName: TableViewCellIdentifiers.loadingCell,
                bundle: nil)
tableView.register(cellNib, forCellReuseIdentifier:
                  TableViewCellIdentifiers.loadingCell)
```

Now you have to come up with some way to let the table view's data source know that the app is currently in a state of downloading data from the server.

The simplest way to do that is to add another boolean flag. If this variable is true, then the app is downloading stuff and the new Loading... cell should be shown; if the variable is false, you show the regular contents of the table view.

- Add a new instance variable:

```
var isLoading = false
```

- Change tableView(numberOfRowsInSection) to:

```
func tableView(_ tableView: UITableView,
               numberOfRowsInSection section: Int) -> Int {
    if isLoading {
        return 1
    } else if !hasSearched {
        . . .
    } else if . . .
```

You've added the if isLoading statement to return 1, because you need a row in order to show a cell.

- Add the following to the top of tableView(cellForRowAt):

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

    if isLoading {
        let cell = tableView.dequeueReusableCell(withIdentifier:
                                                TableViewCellIdentifiers.loadingCell, for: indexPath)

        let spinner = cell.viewWithTag(100) as! UIActivityIndicatorView
        spinner.startAnimating()
        return cell
    }
    else if searchResults.count == 0 {
        . . .
```

You added an if-statement to return an instance of the new Loading... cell. It also looks up the UIActivityIndicatorView by its tag and then tells the spinner to start animating. The rest of the method stays the same.

- Change tableView(willSelectRowAt) to:

```
func tableView(_ tableView: UITableView,
               willSelectRowAt indexPath: IndexPath) -> IndexPath? {
    if searchResults.count == 0 || isLoading {
        return nil
    } else {
        return indexPath
    }
}
```

You added `|| isLoading` to the if-statement. Just like you don't want the users to select the "Nothing Found" cell, you also don't want them to select the "Loading..." cell, so you return `nil` in both cases.

That leaves only one thing to do: you should set `isLoading` to `true` before you make the HTTP request to the iTunes server, and also reload the table view to make the Loading... cell appear.

► Change `searchBarSearchButtonClicked()` to:

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
    if !searchBar.text!.isEmpty {
        searchBar.resignFirstResponder()

        isLoading = true                // add these two lines
        tableView.reloadData()

        // . . . here is the networking code . . .

        isLoading = false              // add this line
        tableView.reloadData()
        return
    }
    . . .
}
```

Before you do the networking request, you set `isLoading` to `true` and reload the table to show the activity indicator.

After the request completes and you have the search results, you set `isLoading` back to `false` and reload the table again to show the `SearchResult` objects.

Makes sense, right? Let's fire up the app and see this in action.

► Run the app and perform a search. While search is taking place the Loading... cell with the spinning activity indicator should appear...

...or should it?!

The sad truth is that there is no spinner to be seen. And in the unlikely event that it does show up for you, it won't be spinning. (Try it with Network Link Conditioner enabled.)

► To show you why, first change `searchBarSearchButtonClicked()` to the following.

You don't have to remove anything from the code, simply comment out everything after the first call to `tableView.reloadData()`.

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
    if !searchBar.text!.isEmpty {
        searchBar.resignFirstResponder()

        isLoading = true
        tableView.reloadData()

        /*
         . . . the networking code (commented out) . . .
        */
    }
}
```

► Run the app and do a search. Now the activity spinner does show up!

So at least you know that part of the code is working fine. But with the networking code enabled the app isn't only totally unresponsive to any input from the user, it also doesn't want to redraw its screen. What's going on here?



The main thread

The CPU (Central Processing Unit) in older iPhone and iPad models has one core, which means it can only do one thing at the time. More recent models have a CPU with two cores, which allows for a whopping two computations to happen simultaneously. Your Mac may have 4 cores.

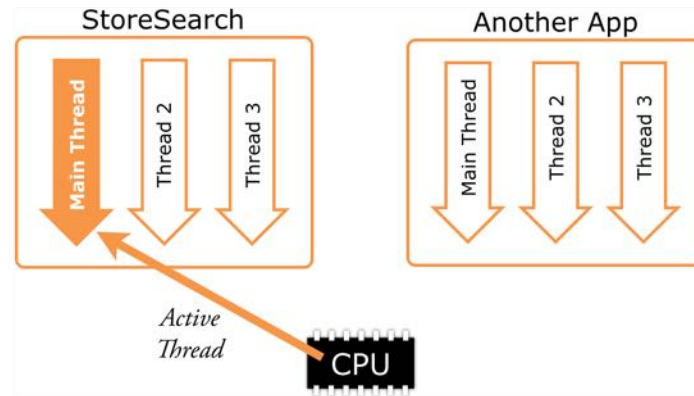
With so few cores available, how come modern computers can have many more applications and other processes running at the same time? (I count 287 active processes on my Mac right now.)

To get around the hardware limitation of having only one or two CPU cores, most computers including the iPhone and iPad use **preemptive multitasking** and **multithreading** to give the illusion that they can do many things at once.

Multitasking is something that happens between different apps. Each app is said to have its own **process** and each process is given a small portion of each second of CPU time to perform its jobs. Then it is temporarily halted, or *pre-empted*, and control is given to the next process.

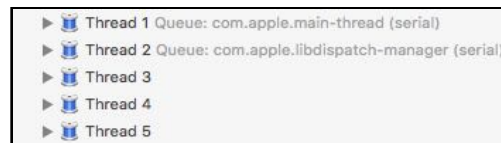
Each process contains one or more **threads**. I just mentioned that each process in turn is given a bit of CPU time to do its work. The process splits up that time among its threads. Each thread typically performs its own work and is as independent as possible from the other threads within that process.

An app can have multiple threads and the CPU switches between them:



If you go into the Xcode debugger and pause the app, the debugger will show you which threads are currently active and what they were doing before you stopped them.

For the StoreSearch app, there were apparently five threads at that time:



Most of these threads are managed by iOS itself and you don't have to worry about them (you may see less or more than five). However, there is one thread that requires special care: the **main thread**. In the image above, that is Thread 1.

The main thread is the app's initial thread and from there all the other threads are spawned. The main thread is responsible for handling user interface events and also for drawing the UI. Most of your app's activities take place on the main thread. Whenever the user taps a button in your app, it is the main thread that performs your action method.

Because it's so important, you should be careful not to hold up, or "block", the main thread. If your action method takes more than a fraction of a second to run, then doing all these computations on the main thread is not a good idea for the reasons you saw earlier.

The app becomes unresponsive because the main thread cannot handle any UI events while you're keeping it busy doing something else – and if the operation takes too long the app may even be killed by the system.

In StoreSearch, you're doing a lengthy network operation on the main thread. It could potentially take many seconds, maybe even minutes, to complete.

After you set the `isLoading` flag to `true`, you tell the `tableView` to reload its data so that the user can see the spinning animation. But that never comes to pass. Telling the table view to reload schedules a “redraw” event, but the main thread gets no chance to handle that event as you immediately start the networking operation, keeping the main thread busy all the time.

This is why I said the current synchronous approach to doing networking was bad: **Never block the main thread**. It’s one of the deadly sins of iOS programming!



Making it asynchronous

To prevent locking up the main thread, any operation that might take a while to complete should be **asynchronous**. That means the operation happens off in the background somewhere and in the mean time the main thread is free to process new events.

That is not to say you should create your own thread. If you’ve programmed on other platforms before you may not think twice about creating new threads, but on iOS that is often not the best solution.

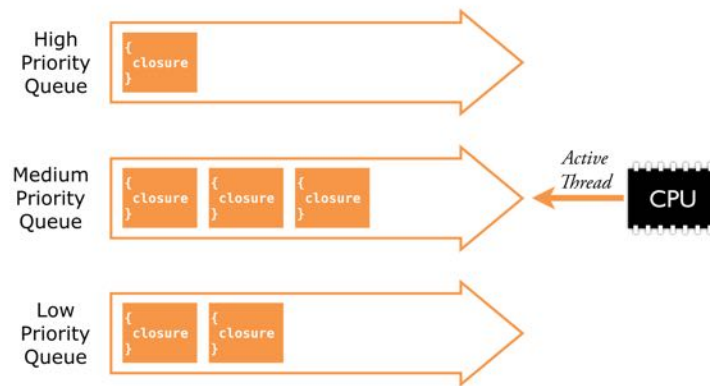
You see, threads are tricky. Not threads per se, but doing things in parallel. Our human minds are very bad at handling the complexity that comes from doing more than one thing at a time – at least when it comes to computations.

I won’t go into too much detail here, but generally you want to avoid the situation where two threads are modifying the same piece of data at the same time. That can lead to very surprising (but not very pleasant!) results.

Rather than making your own threads, iOS has several more convenient ways to start background processes. For this app you’ll be using **queues** and **Grand Central Dispatch** (or GCD). GCD greatly simplifies tasks that require parallel programming. You’ve already briefly played with GCD in the `MyLocations` tutorial, but now you’ll put it to real use.

In short, GCD has a number of queues with different priorities. To perform a job in the background, you put it in a closure and then give that closure to a queue and forget about it. It’s as simple as that.

GCD will pull the closures – or “blocks” as it calls them – from the queues one-by-one and perform their code in the background. Exactly how it does that is not important, you’re only guaranteed it happens on a background thread somewhere. Queues are not exactly the same as threads, but they use threads to do their dirty work.



Queues have a list of closures to perform on a background thread

To make the web service requests asynchronous, you're going to put the networking part from `searchBarSearchButtonClicked()` into a closure and then place that closure on a medium priority queue.

► Change `searchBarSearchButtonClicked()` to the following:

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
    if !searchBar.text!.isEmpty {
        searchBar.resignFirstResponder()

        isLoading = true
        tableView.reloadData()

        hasSearched = true
        searchResults = []

        // 1
        let queue = DispatchQueue.global()
        // 2
        queue.async {
            let url = self.iTunesURL(searchText: searchBar.text!)

            if let jsonString = self.performStoreRequest(with: url),
                let jsonDictionary = self.parse(json: jsonString) {

                self.searchResults = self.parse(dictionary: jsonDictionary)
                self.searchResults.sort(by: <)
                // 3
                print("DONE!")
                return
            }

            print("Error!")
        }
    }
}
```


Here is the new stuff:

1. This gets a reference to the queue. You're using a "global" queue, which is a queue provided by the system. You can also create your own queues, but using a standard queue is fine for this app.
2. Once you have the queue, you can dispatch a closure on it:

```
queue.async {  
    // this is the closure  
}
```

The closure, as usual, is everything between the { and } symbols. Whatever code is in the closure will be put on the queue and be executed asynchronously in the background. After scheduling this closure, the main thread is immediately free to continue. It is no longer blocked.

3. Inside the closure I have removed the code that reloads the table view after the search is done, as well as the error handling code. For now this has been replaced by `print()` statements. There is a good reason for this that we'll get to in a second. First let's try the app again.

► Run the app and do a search. The "Loading..." cell should be visible – complete with animating spinner! After a short while you should see the "DONE!" message appear in the debug pane.

Of course, the Loading... cell sticks around forever because you haven't told it yet to go away.

The reason I removed all the user interface code from the closure is that UIKit has a rule that UI code should *always* be performed on the main thread. This is important!

Accessing the same data from multiple threads can create all sorts of misery, so the designers of UIKit decided that changing the UI from other threads would not be allowed. That means you cannot reload the table view from within this closure, because it runs on a queue that is backed by a thread other than the main thread.

As it happens, there is also a so-called "main queue" that is associated with the main thread. If you need to do anything on the main thread from a background queue, you can simply create a new closure and schedule that on the main queue.

► Replace the line that says `print("DONE!")` with:

```
DispatchQueue.main.async {  
    self.isLoading = false  
    self.tableView.reloadData()  
}
```

With `DispatchQueue.main.async` you can schedule a new closure on the main queue. This new closure sets `isLoading` back to `false` and reloads the table view. Note that

`self` is required because this code sits inside a closure.

► Replace the line that says `print("Error!")` with:

```
DispatchQueue.main.async {  
    self.showNetworkError()  
}
```

You also schedule the call to `showNetworkError()` on the main queue. That method shows a `UIAlertController`, which is UI code and therefore needs to happen on the main thread.

► Try it out. With those changes in place, your networking code no longer occupies the main thread and the app suddenly feels a lot more responsive!

When working with GCD queues you will often see this pattern:

```
let queue = DispatchQueue.global()  
queue.async {  
    // code that needs to run in the background  
  
    DispatchQueue.main.async {  
        // update the user interface  
    }  
}
```

There is also `queue.sync`, without the “a”, which takes the next closure from the queue and performs it in the background, but makes you wait until that closure is done. That can be useful in some cases but most of the time you’ll want to use `queue.async`. No one likes to wait!

► I think with this important improvement the app deserves a new version number, so commit the changes and create a tag for **v0.2**.

You can find the project files for this section under **04 - Async Networking** in the tutorial’s Source Code folder.

URLSession

So far you’ve used the `String(contentsOf, encoding)` method to perform the search on the iTunes web service. That is great for simple apps, but I want to show you another way to do networking that is more powerful.

iOS itself comes with a number of different classes for doing networking, from low-level sockets stuff that is only interesting to really hardcore network programmers, to convenient classes such as `URLSession`.

In this section you’ll replace the existing networking code with the `URLSession` API. That is the API the pros use for building real apps, but don’t worry, it’s not more difficult than what you’ve done before – just more powerful.

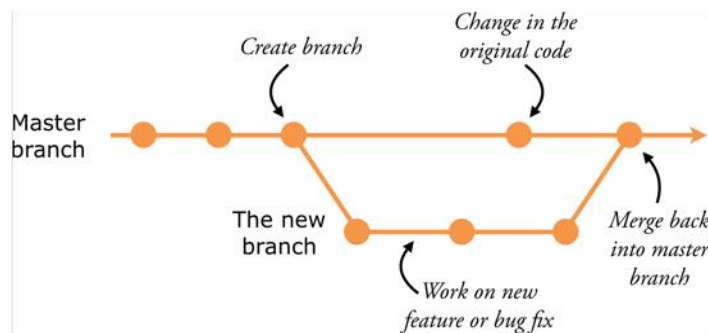
Branch it

Whenever you make a big change to the code, such as replacing all the networking stuff with `NSURLSession`, there is a possibility that you'll mess things up. I certainly do often enough! That's why it's smart to create a so-called Git **branch** first.

The Git repository contains a history of all the app's code, but it can also contain this history along different paths.

You just finished the first version of the networking code and it works pretty well. Now you're going to completely replace that with a – hopefully! – better solution. In doing so, you may want to commit your progress at several points along the way.

What if it turns out that switching to `NSURLSession` wasn't such a good idea after all? Then you'd have to restore the source code to a previous commit from before you started making those changes. In order to avoid this potential mess, you can make a branch instead.

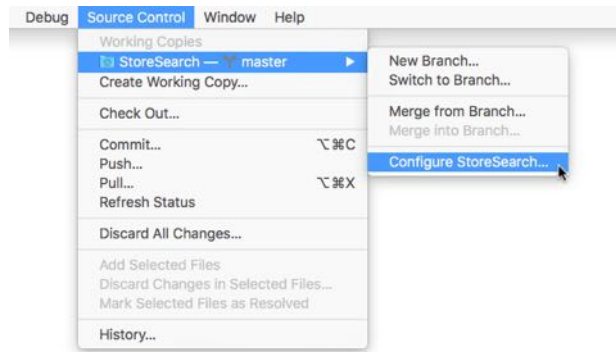


Branches in action

Every time you're about to add a new feature to your code or have a bug to fix, it's a good idea to make a new branch and work on that. When you're done and are satisfied that everything works as it should, merge your changes back into the master branch. Different people use different branching strategies but this is the general principle.

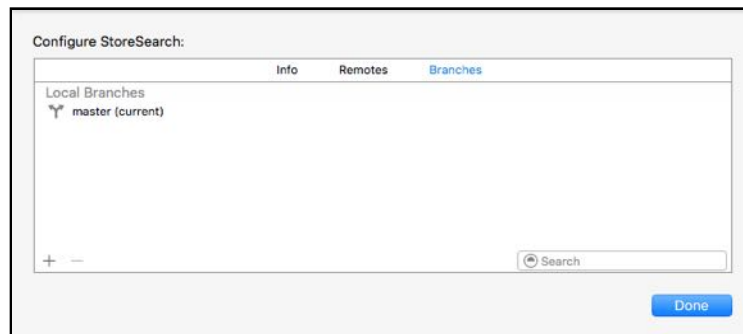
So far you have been committing your changes to the "master" branch. Now you're going to make a new branch, let's call it "urlsession", and commit your changes to that. When you're done with this new feature you will merge everything back into the master branch.

You can find the branches for your repository in the **Source Control** menu:



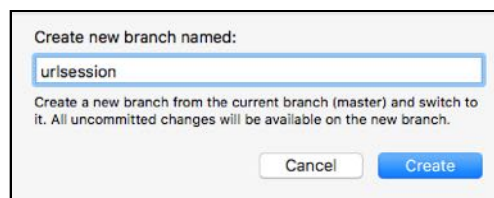
The Source Control branch menu

- Select **StoreSearch – master** (the name of the active branch), and choose **Configure StoreSearch...** to bring up the following panel:



There is currently only one branch in the repository

- Go to the **Branches** tab and click the **+** button at the bottom. In the screen that appears, type **urlsession** for the branch name and click **Create**.



Creating a new branch

When Xcode is done, you'll see that a new "urlsession" branch has been added and that it is made the current one.

This new branch contains the exact same source code and history as the master branch. But from here on out the two paths will diverge – any changes you make happen on the "urlsession" branch only.

Putting URLSession into action

Good, now that you're in a new branch it's safe to experiment with these new APIs.

► First, remove the `performStoreRequest(with)` method from **SearchViewController.swift**. Yup, that's right, you won't be needing it anymore.

Don't be afraid to remove old code. Some developers only comment out the old code but leave it in the project, just in case they may need it again some day.

You don't have to worry about that because you're using source control. Should you really need it, you can always find the old code in the Git history. Besides, if the experiment should fail, you can simply throw away this branch and switch back to the "official" one.

Anyway, on to `URLSession`. This is a closed-based API, meaning that instead of making a delegate, you give it a closure containing the code that should be performed once the response from the server has been received. `URLSession` calls this closure the "completion handler".

► Change `searchBarSearchButtonClicked()` to the following:

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
    if !searchBar.text!.isEmpty {
        searchBar.resignFirstResponder()

        isLoading = true
        tableView.reloadData()

        hasSearched = true
        searchResults = []

        // 1
        let url = iTunesURL(searchText: searchBar.text!)
        // 2
        let session = URLSession.shared
        // 3
        let dataTask = session.dataTask(with: url, completionHandler: {
            data, response, error in
                // 4
                if let error = error {
                    print("Failure! \(error)")
                } else {
                    print("Success! \(response!)")
                }
            })
        // 5
        dataTask.resume()
    }
}
```

This is what the changes do:

1. Create the URL object with the search text like before.

2. Obtain the `URLSession` object. This grabs the standard “shared” session, which uses a default configuration with respect to caching, cookies, and other web stuff.

If you want to use a different configuration – for example, to restrict networking to when Wi-Fi is available but not when there is only cellular access – then you have to create your own `URLSessionConfiguration` and `URLSession` objects. But for this app the default one will be fine.

3. Create a data task. Data tasks are for sending HTTPS GET requests to the server at `url`. The code from the completion handler will be invoked when the data task has received the reply from the server.
4. Inside the closure you’re given three parameters: `data`, `response`, and `error`. These are all optionals so they can be `nil` and have to be unwrapped before you can use them.

If there was a problem, `error` contains an `Error` object describing what went wrong. This happens when the server cannot be reached or the network is down or some other hardware failure.

If `error` is `nil`, the communication with the server succeeded; `response` holds the server’s response code and headers, and `data` contains the actual thing that was sent back from the server, in this case a blob of JSON.

For now you simply use a `print()` to show success or failure.

5. Finally, once you have created the data task, you need to call `resume()` to start it. This sends the request to the server. That all happens on a background thread, so the app is immediately free to continue (`URLSession` is as asynchronous as they come).

With those changes made, you can run the app and see what `URLSession` makes of it.

► Run the app and search for something. After a second or two you should see the debug output say “Success!” followed by a dump of the HTTP response headers.

Excellent!



A brief review of closures

You’ve seen closures a few times now. They are a really powerful feature of Swift and you can expect to be using them all the time when you’re working with Swift code. So it’s good to have at least a basic understanding of how they work.

A closure is simply a piece of source code that you can pass around just like any other type of object. The difference between a closure and regular source code is that the code from the closure does not get performed right away. It is stored in a “closure object” and can be performed at a later point, even more than once.

That’s exactly what `URLSession` does: it keeps hold of the “completion handler” closure and only performs it when a response is received from the web server or when a network error occurs.

A closure typically looks like this:

```
let dataTask = session.dataTask(with: url, completionHandler: {  
    data, response, error in  
    . . . source code . . .  
})
```

The thing behind `completionHandler` inside the `{ }` brackets is the closure. The form of a closure is always:

```
{ parameters in  
  your source code  
}
```

or without parameters:

```
{  
  your source code  
}
```

Just like a method or function, a closure can accept parameters. They are separated from the source code by the “in” keyword. In `URLSession`’s completion handler the parameters are `data`, `response`, and `error`.

Thanks to Swift’s type inference you don’t need to specify the data types of the parameters. However, you could write them out in full if you wanted to:

```
let dataTask = session.dataTask(with: url, completionHandler: {  
    (data: Data?, response: URLResponse?, error: Error?) in  
    . . .  
})
```

Tip: For a parameter without a type annotation, you can Option-click to find out what its type is. This trick works for any symbol in your programs.

If you don’t care about a particular parameter you can substitute it with `_`, the *wildcard* symbol:

```
let dataTask = session.dataTask(with: url, completionHandler: {  
    data, _, error in  
    . . .  
})
```

If a closure is really simple, you can leave out the parameter list altogether and use \$0, \$1, and so on as the parameter names.

```
let dataTask = session.dataTask(with: url, completionHandler: {
    print("My parameters are \"($0), \"($1), \"($2)")
})
```

You wouldn't do that with URLSession's completion handler, though. It's much easier if you know the parameters are called data, response, and error than remembering what \$0, \$1, and \$2 stand for.

If a closure is the last parameter of a method, you can use *trailing* syntax to simplify the code a little:

```
let dataTask = session.dataTask(with: url) {
    data, response, error in
    . . .
}
```

Now the closure sits behind the closing parenthesis, not inside. Many people, myself included, find this more natural to read.

Closures are useful for other things too, such as initializing objects and lazy loading:

```
lazy var dateFormatter: DateFormatter = {
    let formatter = DateFormatter()
    formatter.dateStyle = .medium
    formatter.timeStyle = .short
    return formatter
}()
```

The code to create and initialize the DateFormatter object sits inside a closure. The () at the end causes the closure to be *evaluated* and the returned object is put inside the dateFormatter variable. This is a common trick for placing complex initialization code right next to the variable declaration.

It's no coincidence that closures look a lot like functions. In Swift closures, methods and functions are really all the same thing. For example, you can supply the name of a method or function when a closure is expected, as long as the parameters match:

```
let dataTask = session.dataTask(with: url,
                                completionHandler: myCompletionHandlerMethod)
. . .

func myCompletionHandlerMethod(data: Data?, response: URLResponse?,
                               error: Error?) {
    . . .
}
```

That somewhat negates one of the prime benefits of closures – keeping all the code

in the same place – but there are situations where this is quite useful (the method acts as a “mini” delegate.)

One final thing to be aware of with closures is that they *capture* any variables used inside the closure, including `self`. This can create ownership cycles, often leading to memory leaks. To avoid this, you can supply a *capture list*:

```
let dataTask = session.dataTask(with: url) {  
    [weak self] data, response, error in  
    . . .  
}
```

Whenever you access an instance variable or call a method, you’re implicitly using `self`. Inside a closure, however, Swift requires that you always write “`self.`” in front of the method call or instance variable. This makes it clear that `self` is being captured by the closure:

```
let dataTask = session.dataTask(with: url) {  
    data, response, error in  
    self.callSomeMethod()    // self is required  
}
```

`SearchViewController` doesn’t have to worry about `NSURLSession` capturing `self` because the data task is only short-lived, while the view controller sticks around for as long as the app itself. This ownership cycle is quite harmless. Later on in the tutorial you *will* have to use `[weak self]` with `NSURLSession` or the app might crash and burn!

Note: Swift also has the concept of “no escape” closures. We won’t go into that here, except to mention that no-escape closures don’t capture `self`, so you don’t have to write “`self.`” everywhere. Nice, but you can only use such closures under very specific circumstances!



After a successful request, the app prints the HTTP response from the server. The response object might look something like this:

```
Success! <NSHTTPURLResponse: 0x7f8b19e38d10> { URL: https://  
itunes.apple.com/search?term=metallica&limit=200 } {  
status code: 200, headers {  
    "Cache-Control" = "no-transform, max-age=41";  
    Connection = "keep-alive";  
    "Content-Encoding" = gzip;  
    "Content-Length" = 34254;  
    "Content-Type" = "text/javascript; charset=utf-8";  
    Date = "Fri, 21 Aug 2015 09:53:20 GMT";  
    . . .
```

```
} }
```

If you've done any web development before, this should look familiar. These "HTTP headers" are always the first part of the response from a web server that precedes the actual data you're receiving. The headers give additional information about the communication that just happened.

What you're especially interested in is the status code. The HTTP protocol has defined a number of status codes that tell clients whether the request was successful or not. No doubt you're familiar with 404, web page not found.

The status code you want to see is 200 OK, which indicates success. (Wikipedia has the complete list of codes, wikipedia.org/wiki/List_of_HTTP_status_codes.)

To make the error handling of the app a bit more robust, let's check to make sure the HTTP response code really was 200. If not, something has gone wrong and we can't assume that data contains the JSON we're after.

► Change the contents of the completionHandler to:

```
if let error = error {
    print("Failure! \(error)")
} else if let httpResponse = response as? HTTPURLResponse,
           httpResponse.statusCode == 200 {
    print("Success! \(data!)")
} else {
    print("Failure! \(response)")
}
```

The response parameter has the data type `URLResponse` but that doesn't have a property for the status code. Because you're using the HTTP protocol, what you've really received is an `HTTPURLResponse` object, a subclass of `URLResponse`.

So first you cast it to the proper type and then look at its `statusCode` property. Only if it is 200 you'll consider the job a success.

Notice the use of the comma inside the `if let` statement to combine these checks into a single line. You could also have written it with a second `if`, but I find that harder to read:

```
} else if let httpResponse = response as? HTTPURLResponse {
    if httpResponse.statusCode == 200 {
        print("Success! \(data!)")
    }
}
```

Whenever you need to unwrap an optional and also check the value of that optional, using `if let ..., ...` is the nicest way to do that.

► Run the app and search for something. You should now see something like:

```
Success! <0a0a0a7b 0a202272 6573756c 74436f75 6e74223a 3230302c 0a202272
6573756c 7473223a 205b0a7b 22777261 70706572 54797065 223a2274 7261636b
```

```
222c2022 6b696e64 223a2266 65617475 72652d6d 6f766965 222c2022 74726163
6b496422 3a353338 38303035 39382c20 22617274 6973744e 616d6522 3a224c69
. . .
```

This is the data object with the JSON search results. The JSON is really text but because data is in the form of a Data object it prints out its contents as binary (hexadecimal to be precise). In a minute you'll turn this into real JSON objects.

It's always a good idea to actually test your error handling code, so let's first fake an error and get that out of the way.

► In `iTunesURL(searchText)`, change the string to:

```
"https://itunes.apple.com/searchL0L?term=%@&limit=200"
```

Here I've changed the endpoint from `search` to `searchL0L`. It doesn't really matter what you type there, as long as it's something that cannot possibly exist on the iTunes server.

► Run the app again. Now a search should respond with something like this:

```
Failure! <NSHTTPURLResponse: 0x7ff76b42d4b0> { URL: https://
itunes.apple.com/searchL0L?term=metallica&limit=200 } {
status code: 404, headers {
    Connection = "keep-alive";
    "Content-Length" = 207;
    "Content-Type" = "text/html; charset=iso-8859-1";
    . . .
} }
```

As you can see, the status code is now 404 – there is no `searchL0L` page – and the app correctly considers this a failure. That's a good thing too, because data now contains the following:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL /searchL0L was not found on this server.</p>
</body></html>
```

That is definitely not JSON but HTML. If you tried to convert that into JSON objects, you'd fail horribly.

Great, so the error handling works. Let's add JSON parsing to the code.

► First, put `iTunesURL(searchText)` back to the way it was (⌘+Z to undo).

► Then change `parse(json)` to the following:

```
func parse(json data: Data) -> [String: Any]? {
    do {
```

```
        return try JSONSerialization.jsonObject(with: data, options: [])
                                              as? [String: Any]
    } catch {
        print("JSON Error: \(error)")
        return nil
    }
}
```

You removed the guard statement and changed the parameter from `String` to `Data`.

Previously this method took a `String` object and converted it into a `Data` object that it passed to `JSONSerialization.jsonObject(...)`. Now you already have the JSON text in a `Data` object, so you no longer have to bother with the string.

The app is not doing anything yet with the search results, but you already wrote all the code you need for that before, so let's put it in the closure.

► In the completion handler, replace the `print("Success! \(data)")` line with:

```
if let data = data, let jsonDictionary = self.parse(json: data) {
    self.searchResults = self.parse(dictionary: jsonDictionary)
    self.searchResults.sort(by: <)

    DispatchQueue.main.async {
        self.isLoading = false
        self.tableView.reloadData()
    }
    return
}
```

This unwraps the optional object from the `data` parameter and gives it to `parse(json)` to convert it into a dictionary. Then it calls `parse(dictionary)` to turn the dictionary's contents into `SearchResult` objects, just like you did before. Finally, you sort the results and put everything into the table view. This should look very familiar.

It's important to realize that the completion handler closure won't be performed on the main thread. Because `URLSession` does all the networking asynchronously, it will also call the completion handler on a background thread.

Parsing the JSON and sorting the list of search results could potentially take a while (not seconds but possibly long enough to be noticeable). You don't want to block the main thread while that is happening, so it's preferable that this happens in the background too.

But when the time comes to update the UI, you need to switch back to the main thread. That's the rules. That's why you wrap the reloading of the table view into `DispatchQueue.main.async` on the main queue.

If you forget to do this, your app may still appear to work. That's the insidious thing about working with multiple threads. However, it may also crash in all kinds of mysterious ways. So remember, UI stuff should always happen on the main thread.

Write it on a Post-It note and stick it to your screen!

➤ Run the app. The search should work again. You have successfully replaced the old networking code with URLSession!

Tip: To tell whether a particular piece of code is being run on the main thread, add the following code snippet:

```
print("On main thread? " + (Thread.current.isMainThread ? "Yes" : "No"))
```

Go ahead, paste this at the top of the completionHandler closure and see what it says.

Of course, the official framework documentation should be your first stop. Usually when a method takes a closure the docs mention whether it is performed on the main thread or not. But if you're not sure, or just can't find it in the docs, add the above print() and be enlightened.

➤ At the very bottom of the completion handler closure, below the if-statements, add the following:

```
DispatchQueue.main.async {  
    self.hasSearched = false  
    self.isLoading = false  
    self.tableView.reloadData()  
    self.showNetworkError()  
}
```

The code gets here if something went wrong. You call showNetworkError() to let the user know about the problem.

Note that you do tableView.reloadData() here too, because the contents of the table view need to be refreshed to get rid of the Loading... indicator. And of course, all this happens on the main thread.

Exercise. Why doesn't the error alert show up on success? After all, the above piece of code sits at the bottom of the closure, so doesn't it always get executed? ■

Answer: Upon success, the return statement exits the closure after the search results get displayed in the table view. So in that case execution never reaches the bottom of the closure.

➤ Fake an error situation to test that the error handling code really works.

Testing errors is not a luxury! The last thing you want is your app to crash when a networking error occurs because of faulty error handling code. I've worked on codebases where it was obvious the previous developer never bothered to verify that the app was able to recover from errors. (That's probably why they were the *previous* developers.)

Things will go wrong in the wild and your app better be prepared to deal with it. As

the MythBusters say, “failure is always an option”.

Does the error handling code work? Great! Time to add some new networking features to the app.

► This is a good time to commit your changes. Remember, this commit only happens on the “urlsession” branch, not on the master branch.

Canceling operations

What happens when a search takes very long and the user already starts a second search when the first one is still going? The app doesn’t disable the search bar so it’s possible for the user to pull this off. When dealing with networking – or any asynchronous process, really – you have to think these kinds of situations through.

There is no way to predict what happens, but it will most likely be a strange experience for the user. She might see the results from her first search, which she is no longer expecting (confusing!), only to be replaced by the results of the second search a few seconds later.

But there is no guarantee the first search completes before the second, so the results from search 2 may arrive first and then get overwritten by the results from search 1, which is definitely not what the user wanted to see either.

Because you’re no longer blocking the main thread, the UI always accepts user input, and you cannot assume the user to sit still and wait until the request is done.

You can usually fix this dilemma in one of two ways:

1. Disable all controls. The user cannot tap anything while the operation is taking place. This does not mean you’re blocking the main thread; you’re just making sure the user cannot mess up the order of things.
2. Cancel the on-going request when the user starts a new one.

For this app you’re going to pick the second solution because it makes for a nicer user experience. Every time the user performs a new search you cancel the previous request. `NSURLSession` makes this easy: data tasks have a `cancel()` method.

When you created the data task, you were given a `NSURLSessionDataTask` object, and you placed this into the local constant named `dataTask`. Cancelling the task, however, needs to happen the *next* time `searchBarSearchButtonClicked()` is called.

Storing the `NSURLSessionDataTask` object into a local variable isn’t good enough anymore; you need to keep that reference beyond the scope of this method. In other words, you have to put it into an instance variable.

► Add the following instance variable to **SearchViewController.swift**:

```
var dataTask: URLSessionDataTask?
```

This is an optional because you won't have a data task yet until the user performs a search.

► Inside `searchBarSearchButtonClicked()`, remove `let` from the line that creates the new data task object:

```
dataTask = session.dataTask(with: url, completionHandler: {
```

You've removed the `let` keyword because `dataTask` should no longer be a local; it now refers to the instance variable.

► At the bottom of the method, add a question mark to the line that starts the task:

```
dataTask?.resume()
```

Because `dataTask` is an optional, you have to unwrap the optional somehow before you can use it. Here you're using optional chaining.

► Finally, near the top of the method before you set `isLoading` to `true`, add:

```
dataTask?.cancel()
```

If there was an active data task this cancels it, making sure that no old searches can ever get in the way of the new search.

Thanks to the optional chaining, if no search was done yet and `dataTask` is still `nil`, this simply ignores the call to `cancel()`. You could also unwrap the optional with `if let`, but using the question mark is shorter and just as safe.

Exercise. Why can't you write `dataTask!.cancel()` to unwrap the optional? ■

Answer: If an optional is `nil`, using `!` will crash the app. You're only supposed to use `!` to unwrap an optional when you're sure it won't be `nil`. But the very first time the user types something into the search bar, `dataTask` will still be `nil` and using `!` is not a good idea.

► Test the app with and without this call to `dataTask.cancel()` to experience the difference.

Use the Network Link Conditioner preferences pane to delay each query by a few seconds so it's easier to get two requests running at the same time.

Hmm... you may have noticed something odd. When the data task gets cancelled, you get the error popup and the Debug pane says:

```
Failure! Error Domain=NSURLErrorDomain Code=-999 "cancelled" UserInfo= ...  
{NSErrorFailingURLKey=https://itunes.apple.com/search?term=monkeys&  
limit=200, NSLocalizedDescription=cancelled, NSErrorFailingURLStringKey=
```

As it turns out, when a data task gets cancelled its completion handler is still

invoked but with an Error object that has error code -999. That's what caused the error alert to pop up.

You'll have to make the error handler a little smarter to ignore code -999. After all, the user cancelling the previous search is no cause for panic.

► In the completionHandler, change the if let error section to:

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

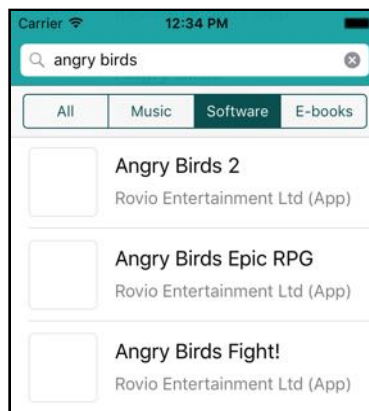
This simply ends the closure when there is an error with code -999. The rest of the closure gets skipped.

► If you're satisfied it works, commit the changes to the repository.

Note: Maybe you don't think it's worth making a commit when you've only changed a few lines, but many small commits are often better than a few big ones. Each time you fix a bug or add a new feature is a good time to commit.

Searching different categories

The iTunes store has a vast collection of products and each search returns at most 200 items. It can be hard to find what you're looking for by name alone, so you'll add a control to the screen that lets users pick the category they want to search in. It looks like this:



Searching in the Software category

This type of control is called a **segmented control** and is used to pick one option out of multiple choices.

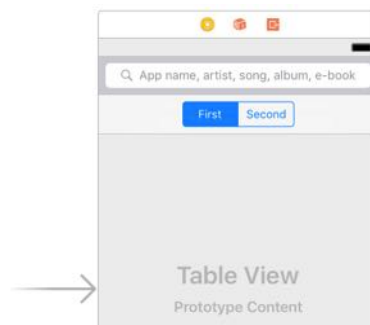
► Open the storyboard. Drag a new **Navigation Bar** into the view and put it below the Search Bar. You're using the Navigation Bar purely for decorative purposes, as a container for the segmented control.

Make sure the Navigation Bar doesn't get added inside the Table View. It may be easiest to drag it from the Object Library directly into the outline pane and drop it below the Search Bar. Then change its Y-position to 64.

➤ With the Navigation Bar selected, open the **Pin menu** and pin its **top**, **left**, and **right** sides.

➤ Drag a new **Segmented Control** from the Object Library on top of the Navigation Bar's title (so it will replace the title).

The design now looks like this:



The Segmented Control sits in a Navigation Bar below the Search Bar

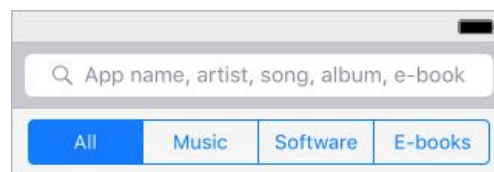
➤ Select the Segmented Control. Set its **Width** to 300 points (make sure you change the width of the entire control, not of the individual segments).

➤ In the **Attributes inspector**, set the number of segments to 4.

➤ Change the title of the first segment to **All**. Then select the second segment and set its title to **Music**. The title for the third segment should be **Software** and the fourth segment is **E-books**.

You can change the segment title by double-clicking inside the segment or inside the Attributes inspector.

The scene should look like this now:



The finished Segmented Control

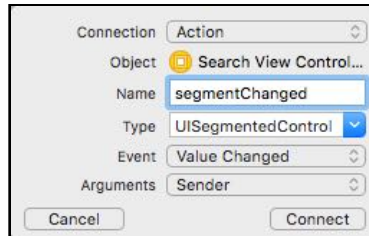
Next you'll add a new outlet and action method for the Segmented Control. This is a good opportunity to practice using the Assistant editor.

➤ Press **Option+⌘+Enter** to open the Assistant editor and then Ctrl-drag from the

Segmented Control into the view controller source code to add the new outlet:

```
@IBOutlet weak var segmentedControl: UISegmentedControl!
```

To add the action method you can also use the Assistant editor. Ctrl-drag from the Segmented Control into the source code again, but this time choose:



Adding an action method for the segmented control

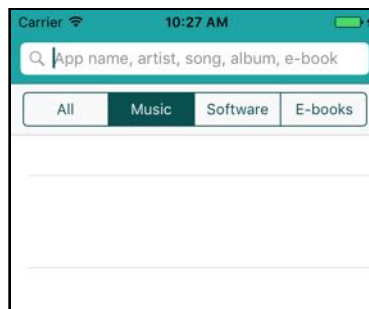
- Connection: **Action**
- Name: **segmentChanged**
- Type: **UISegmentedControl**
- Event: Value Changed
- Arguments: Sender

➤ Press **Connect** to add the action method. Also add a `print()` statement to it:

```
@IBAction func segmentChanged(_ sender: UISegmentedControl) {  
    print("Segment changed: \(sender.selectedSegmentIndex)")  
}
```

Type **⌘+Enter** (without Option) to close the Assistant editor again. These are very handy keyboard shortcuts to remember.

➤ Run the app to make sure everything still works. Tapping a segment should log a number (the index of that segment) to the debug pane.



The segmented control in action

Notice that the first row of the table view is partially obscured again. Because you placed a navigation bar below the search bar, you need to add another 44 points to the table view's content inset.

➤ Change that line in `viewDidLoad()` to:

```
tableView.contentInset = UIEdgeInsets(top: 108, left: 0, . . .
```

You will be using the segmented control in two ways. First of all, it determines what sort of products the app will search for. Second, if you have already performed a search and you tap on one of the other segment buttons, the app will search again for that new product category.

That means a search can now be triggered by two different events: tapping the Search button on the keyboard and tapping in the Segmented Control.

➤ Rename the `searchBarSearchButtonClicked()` method to `performSearch()` and also remove the `searchBar` parameter.

You're doing this to put the search logic into a separate method that can be invoked from more than one place. Removing `searchBar` as the parameter of this method is no problem because there is also an `@IBOutlet` instance variable with that name and `performSearch()` will simply use that.

➤ Now add a new version of `searchBarSearchButtonClicked()` back into the source code:

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {  
    performSearch()  
}
```

➤ Also replace the `segmentChanged()` action method with:

```
@IBAction func segmentChanged(_ sender: UISegmentedControl) {  
    performSearch()  
}
```

➤ Run the app and verify that searching still works. When you tap on the different segments the search should be performed again as well.

Note: The second time you search for the same thing the app may return results very quickly. The networking layer is now returning a *cached* response so it doesn't have to download the whole thing again, which is usually a performance gain on mobile devices. (There is an API to turn off this caching behavior if that makes sense for your app.)

You still have to tell the app to use the category from the selected segment for the search. You've already seen that you can get the index of the selected segment with the `selectedSegmentIndex` property. This returns an `Int` value (0, 1, 2, or 3).

► Change the `iTunesURL(searchText)` method so that it accepts this `Int` as a parameter and then builds up the request URL accordingly:

```
func iTunesURL(searchText: String, category: Int) -> URL {
    let entityName: String
    switch category {
        case 1: entityName = "musicTrack"
        case 2: entityName = "software"
        case 3: entityName = "ebook"
        default: entityName = ""
    }

    let escapedSearchText = searchText.addingPercentEncoding(
        withAllowedCharacters: CharacterSet.urlQueryAllowed)!

    let urlString = String(format:
        "https://itunes.apple.com/search?term=%@&limit=200&entity=%@",
        escapedSearchText, entityName)

    let url = URL(string: urlString)
    return url!
}
```

This first turns the category index from a number into a string, `entityName`. (Note that the category index is passed to the method as a new parameter.)

Then it puts this string behind the `&entity=` parameter in the URL. For the “All” category, the entity value is empty but for the other categories it is “musicTrack”, “software”, and “ebook”, respectively.

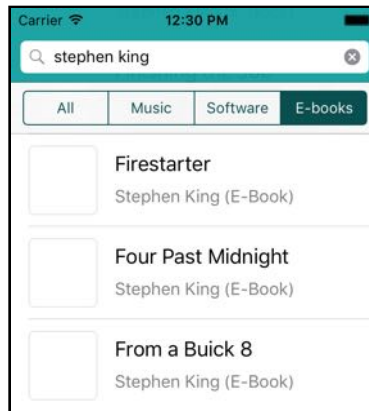
► In the `performSearch()` method, change the line that used to call `iTunesURL()` into the following:

```
let url = self.iTunesURL(searchText: searchBar.text!,
    category: segmentedControl.selectedSegmentIndex)
```

And that should do it.

Note: You could have used `segmentedControl.selectedSegmentIndex` directly inside `iTunesURL(...)` instead of passing the category index as a parameter. Using the parameter is the better design, though. It makes it possible to reuse the same method with a different type of control, should you decide that a Segmented Control isn’t really the right component for this app. It is always a good idea to make methods as independent from each other as possible.

► Run the app and search for “stephen king”. In the All category that gives results for anything from movies to podcasts to audio books. In the Music category it matches mostly artists with the word “King” in their name. There doesn’t seem to be a lot of Stephen King-related software, but in the E-Books category you finally find some of his novels.



You can now limit the search to just e-books

This finalizes the UI design of the main screen. This is as good a point as any to replace the empty white launch file from the template.

- Remove the **LaunchScreen.storyboard** file from the project.
- In the **Project Settings** screen, under **App Icons and Launch Images**, change **Launch Screen File** to **Main.storyboard**.

Now when the app starts up it uses the initial view controller from the storyboard as the launch image. Also verify that the app works properly on the iPad simulator and the larger iPhone 6s, 7, and Plus models.

- Commit the changes and get ready for some more networking!

Downloading the artwork images

The JSON search results contain a number of URLs to images and you put two of those – `artworkSmallURL` and `artworkLargeURL` – into the `SearchResult` object. Now you are going to download these images over the internet and put them into the table view cells.

Downloading images, just like using a web service, is simply a matter of doing an HTTP GET request to a server that is connected to the internet. An example of such a URL is:

<http://a5.mzstatic.com/us/r30/Music/5c/16/8d/mzi.ezpjahaj.100x100-75.jpg>

Click that link and it will open the picture in a new web browser window. The server where this picture is stored is not `itunes.apple.com` but `a5.mzstatic.com`, but that doesn't matter anything to the app.

As long as it has a valid URL, the app will just go fetch the file at that location, no matter where it is and no matter what kind of file that is.

There are various ways that you can download files from the internet. You're going to use `NSURLSession` and write a handy `UIImageView` extension to make this really

convenient. Of course, you'll be downloading these images asynchronously!

First, you will move the logic for configuring the contents of the table view cells into the `SearchResultCell` class. That's a better place for it. Logic related to an object should live inside that object as much as possible, not somewhere else.

Many developers have a tendency to stuff everything into their view controllers, but if you can move some of the logic into other objects that makes for a much cleaner program.

➤ Add the following method to **`SearchResultCell.swift`**:

```
func configure(for searchResult: SearchResult) {
    nameLabel.text = searchResult.name

    if searchResult.artistName.isEmpty {
        artistNameLabel.text = "Unknown"
    } else {
        artistNameLabel.text = String(format: "%@ (%@)",
                                       searchResult.artistName, kindForDisplay(searchResult.kind))
    }
}
```

This is the same as what you used to do in `tableView(cellForRowAt)`. The only problem is that this class doesn't have the `kindForDisplay()` method yet.

➤ Cut the `kindForDisplay()` method out of **`SearchViewController.swift`** and paste it into **`SearchResultCell.swift`**.

It's easy to move this method from one class to another because it doesn't depend on any instance variables. It is completely self-contained. You should strive to write your methods in that fashion as much as possible.

➤ Finally, change `tableView(cellForRowAt)` to the following:

```
func tableView(_ tableView: UITableView,
               cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    if isLoading {
        . . .
    } else if searchResults.count == 0 {
        . . .
    } else {
        let cell = tableView.dequeueReusableCell(withIdentifier: . . .)

        let searchResult = searchResults[indexPath.row]
        cell.configure(for: searchResult)           // change this line
        return cell
    }
}
```

This small refactoring of moving some code from one class (`SearchViewController`) into another (`SearchResultCell`) was necessary to make the next bit work right.

In hindsight, it makes more sense to do this sort of thing in `SearchResultCell`

anyway, but until now it did not really matter. Don't be afraid to refactor your code! (Remember, if you screw up you can always go back to the Git history.)

► Run the app to make sure everything still works as before.

OK, here comes the cool part. You will now make an extension for UIImageView that loads the image and automatically puts it into the image view on the table view cell with just one line of code.

As you know, an extension can be used to extend the functionality of an existing class without having to subclass it. This works even for classes from the system frameworks.

UIImageView doesn't have built-in support for downloading images, but this is a very common thing to do in apps. It's great that you can simply plug in your own extension – and from then on every UIImageView in your app has this new ability.

► Add a new file to the project using the **Swift File** template, and name it **UIImageView+DownloadImage.swift**.

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

```
import UIKit

extension UIImageView {
    func loadImage(url: URL) -> URLSessionDownloadTask {
        let session = URLSession.shared
        // 1
        let downloadTask = session.downloadTask(with: url,
            completionHandler: { [weak self] url, response, error in
                // 2
                if error == nil, let url = url,
                    let data = try? Data(contentsOf: url), // 3
                    let image = UIImage(data: data) {
                    // 4
                    DispatchQueue.main.async {
                        if let strongSelf = self {
                            strongSelf.image = image
                        }
                    }
                }
            })
        // 5
        downloadTask.resume()
        return downloadTask
    }
}
```

This should look very similar to what you did before with URLSession, but there are some differences:

1. After obtaining a reference to the shared URLSession, you create a download task. This is similar to a data task but it saves the downloaded file to a temporary location on disk instead of keeping it in memory.

2. Inside the completion handler for the download task you're given a URL where you can find the downloaded file (this URL points to a local file rather than an internet address). Of course, you must also check that error is nil before you continue.
3. With this local URL you can load the file into a Data object and then make an image from that. It's possible that constructing the UIImage fails, when what you downloaded was not a valid image but a 404 page or something else unexpected. As you can tell, when dealing with networking code you need to check for errors every step of the way!
4. Once you have the image you can put it into the UIImageView's image property. Because this is UI code you need to do this on the main thread.

Here's the tricky thing: it is theoretically possible that the UIImageView no longer exists by the time the image arrives from the server. After all, it may take a few seconds and the user can still navigate through the app in the mean time.

That won't happen in this part of the app because the image view is part of a table view cell and they get recycled but not thrown away. But later in the tutorial you'll use this same code to load an image on a screen that may be closed while the image file is still downloading. In that case you don't want to set the image if the UIImageView is not visible anymore.

That's why the capture list for this closure includes [weak self], where self now refers to the UIImageView. Inside the DispatchQueue.main.async you need to check whether "self" still exists; if not, then there is no more UIImageView to set the image on.

5. After creating the download task you call resume() to start it, and then return the URLSessionDownloadTask object to the caller. Why return it? That gives the app the opportunity to call cancel() on the download task. You'll see how that works in a minute.

And that's all you need to do. From now on you can call loadImage(url) on any UIImageView object in your project. Cool, huh!

Note: Swift lets you combine multiple if let statements into a single line, like you did above:

```
if error == nil, let url = ..., let data = ..., let image = ... {
```

There are three optionals being unwrapped here: 1) url, 2) the result from Data(contentsOf), and 3) the result from UIImage(data).

You can write this as three separate if let statements, and one for if error == nil, but I find that having everything inside a single if-statement is easier to read than many nested if-statements spread over several lines.

➤ Switch to **SearchResultCell.swift** and add the following lines to the bottom of `configure(for):`

```
artworkImageView.image = UIImage(named: "Placeholder")
if let smallURL = URL(string: searchResult.artworkSmallURL) {
    downloadTask = artworkImageView.loadImage(url: smallURL)
}
```

This tells the `UIImageView` to load the image from `artworkSmallURL` and to place it in the cell's image view. While the real artwork is downloading the image view displays a placeholder image (the same one from the nib for this cell).

The `URLSessionDownloadTask` object returned by `loadImage(url)` is placed in a new instance variable, `downloadTask`. You still need to add this instance variable:

```
var downloadTask: URLSessionDownloadTask?
```

➤ Run the app and look at that... error message?!

The Debug pane now says something like this:

```
App Transport Security has blocked a cleartext HTTP (http://) resource
load since it is insecure.
```

Remember how I said that as of iOS 9 you can no longer download files over HTTP but that you always need to use HTTPS? Well, as it happens the iTunes web service gives you image URLs that start with `http://`, not with `https://`. The server that hosts those images apparently does not speak HTTPS at all, so `URLSession` cannot use a secure connection and therefore the request fails.

Fortunately, you can add a key to the app's `Info.plist` to bypass this App Transport Security feature, allowing you to use plain `http://` URLs.

➤ Open **Info.plist** and add a new row. Give it the key **NSAppTransportSecurity** (or choose **App Transport Security Settings** from the list).

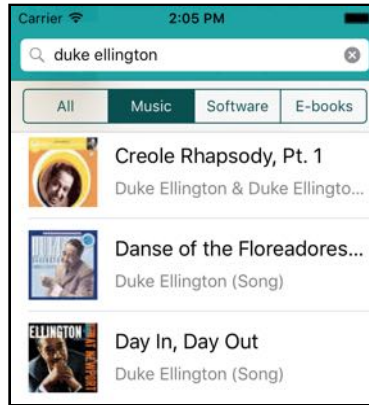
➤ Make sure the Type is a Dictionary.

➤ Add a new key inside that dictionary named **NSAllowsArbitraryLoads** (or choose **Allow Arbitrary Loads** from the list). Make this a Boolean and set it to YES.

| | | |
|--------------------------|------------|----------|
| ▼ NSAppTransportSecurity | Dictionary | (1 item) |
| NSAllowsArbitraryLoads | Boolean | YES |

Overriding App Transport Security

➤ Run the app and look at those icons!



The app now downloads the album artwork



App Transport Security

You're only supposed to bypass App Transport Security if there is absolutely no way you can make the app work over HTTPS. If you're making an app that talks to a server you control, then the best thing to do is to enable HTTPS on the server, not disable HTTPS in the app.

The Info.plist setting is only intended for when you need to communicate with other people's servers that do not speak HTTPS. Obviously, in that case the app should not send sensitive data to those servers! Unprotected HTTP should only be used for downloading publicly accessible data, such as images.

When you set the key `NSAllowsArbitraryLoads` to YES, the app can use *any* URL that starts with `http://`, regardless of the domain. To allow HTTP on specific domains only, set `NSAllowsArbitraryLoads` to NO and add a new dictionary named `NSExceptionDomains`. Inside it you add a new dictionary for each domain.

For example, the iTunes web service appears to host all its preview images on the website `mzstatic.com`. You could configure Info.plist as follows:

| | | |
|------------------------------------|------------|-----------|
| ▼ NSAppTransportSecurity | Dictionary | (2 items) |
| NSAllowsArbitraryLoads | Boolean | NO |
| ▼ NSExceptionDomains | Dictionary | (1 item) |
| ▼ mzstatic.com | Dictionary | (2 items) |
| NSExceptionAllowsInsecureHTTPLoads | Boolean | YES |
| NSIncludesSubdomains | Boolean | YES |

Now the app only allows `http://` requests from `mzstatic.com` and any of its subdomains, but requires `https://` URLs for any other domains.

Note that if you add these kinds of exceptions in your Info.plist, you'll have to explain to Apple why your app needs to make unsecure http:// connections. Without a good reason, they may reject your app from the App Store!



These images already look pretty sweet, but you're not quite done yet. Remember that table view cells can be reused, so it's theoretically possible that you're scrolling through the table and some cell is about to be reused while its previous image is still loading.

You no longer need that image so you should really cancel the pending download. Table view cells have a special method named `prepareForReuse()` that is ideal for this.

► Add the following method to **SearchResultCell.swift**:

```
override func prepareForReuse() {  
    super.prepareForReuse()  
  
    downloadTask?.cancel()  
    downloadTask = nil  
}
```

Here you cancel any image download that is still in progress.

Exercise. Put a `print()` in the `prepareForReuse()` method and see if you can trigger it. ■

On a decent Wi-Fi connection, loading the images is very fast. You almost cannot see that it happens, even if you scroll quickly. It also helps that the image files are small (only 60 by 60 pixels) and that the iTunes servers are fast.

That is key to having a snappy app: don't download more data than you need to.



Caching

Depending on what you searched for, you may have noticed that many of the images were the same. For example, my search for Duke Ellington's music had many identical album covers in the search results.

`URLSession` is smart enough not to download identical images – or at least images with identical URLs – twice. That principle is called **caching** and it's very important

on mobile devices.

Mobile developers are always trying to optimize their apps to do as little as possible. If you can download something once and then use it over and over, that's a lot more efficient than re-downloading it all the time.

There's more than just images that you can cache. You can also cache the results of big computations, for example. Or views, as you have been doing in the previous tutorials, probably without even realizing it. When you use the principle of lazy loading, you delay the creation of an object until you need it and then you cache it for the next time.

Cached data does not stick around forever. When your app gets a memory warning, it's a good idea to remove any cached data that you don't need right away. That means you will have to reload that data when you need it again later but that's the price you have to pay. (For `URLSession` this is completely automatic, so that takes another burden off your shoulders.)

Some caches are in-memory only where the cached data stays in the computer's working memory, but it is also possible to cache the data in files on the disk. Your app even has a special directory for it, `Library/Caches`.

The caching policy used by StoreSearch is very simple – it uses the default settings. But you can configure `URLSession` to be much more advanced. Look into `URLCache` and `URLSessionConfiguration` to learn more.



Merging the branch

This concludes the section on talking to the web service and downloading images. Later on you'll tweak the web service requests a bit more (to include the user's language and country) but for now you're done with this feature.

I hope you got a good glimpse of what is possible with web services and how easy it is to build this into your apps with a great library such as `URLSession`.

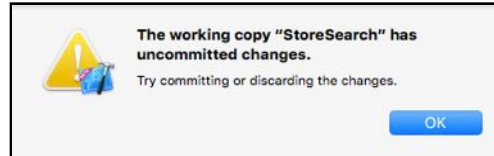
➤ Commit these latest changes to the repository.

Now that you've completed a feature, you can merge this temporary branch back into the master branch. To do that, you first have to return to the master branch.

Merging is possible to do in Xcode but it's not always the best experience. I will first explain how to merge the branch using Xcode. If it doesn't work and Xcode keeps messing up your files, then skip ahead to the command line instructions. (It may be a good idea to make a backup copy of your project folder first.)

► Open the **Source Control** menu. Go to the **StoreSearch – urlsession** submenu and choose **Switch to Branch**.

It is possible that you get this message:



The message that appears when you have uncommitted changes

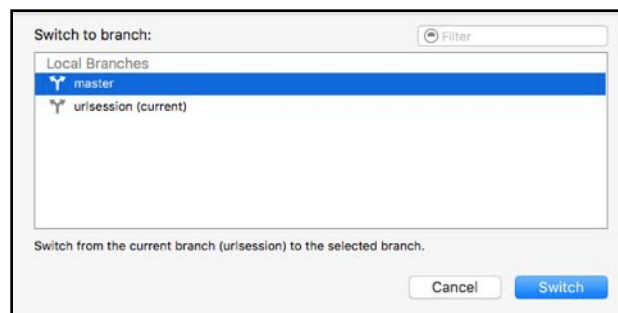
That means changes were made to the source code, the storyboard, or any other files that are being tracked by Git since your last commit. You may have done that yourself – often just opening a storyboard causes it to be modified – but sometimes Xcode itself decides to change things like the .xcodeproj file. It's also possible that you did not commit some of the files that have changes (possibly because it's a file that you don't care about).

If that happens you have two choices:

1. Commit those changes. Simply do a new commit from the Source Control menu.
2. Discard the changes. Choose **Source Control** → **Discard All Changes**.

Then try **Switch to Branch** again.

You should see a dialog that lets you pick the branch to switch to. Select **master** and click **Switch**:

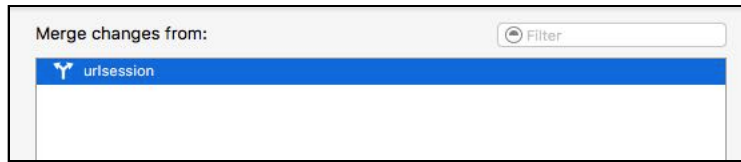


Choosing the branch to switch to

After a few moments, Xcode will have switched back to the master branch. Verify this in the **Source Control** menu: it should now say **StoreSearch – master**. In the History window you should see only the commits up to "asynchronous networking", but none of the URLSession stuff.

► To merge the "urlsession" branch into the master branch, go to **Source Control** → **StoreSearch – master** → **Merge from Branch**.

(If you get the “The working copy has uncommitted changes” warning again, then first choose Discard All Changes. You may have to do this more than once.)



Choosing the branch to merge from

Choose the **urlsession** branch and click **Merge**. This brings up a preview pane that lets you review the changes that will be made. Click **Merge** again.

Now the master branch is up-to-date with the networking changes. If you want to, you can remove the “urlsession” branch or you can keep it and do more work on it later.

Just in case Xcode didn’t want to cooperate, here is how you’d do it from the command line.

- First close Xcode. You don’t want to do any of this while Xcode still has the project open. That’s just asking for trouble.
- Open a Terminal, cd to the StoreSearch folder, and type the following commands:

```
git stash
```

This moves any unsaved files out of the way (it doesn’t have anything to do with facial hair). This is similar to doing Discard All Changes from the Xcode menu, although stashed changes are preserved in a temporary location, not thrown away.

```
git checkout master
```

This switches the current branch back to the master branch.

```
git merge urlsession
```

This merges the changes from the “urlsession” branch back into the master branch. If you get an error message at this point, then simply do `git stash` again and repeat the `git merge` command.

(By the way, you don’t really need to keep those stashed files around, so if you want to remove them from your repository, you can do `git stash drop`. If you stashed twice, you also need to drop twice.)

- Open the project again in Xcode. Now you’re back at the master branch and it also has the latest networking changes.
- Build and run to see if everything still works.

Git is a pretty awesome tool but it takes a while to get familiar with. Unfortunately, Xcode's support for things like merges is still spotty and you're better off using the command line for the more advanced commands. It's well worth learning!

You can find the files for the app up to this point under **05 - URLSession** in the tutorial's Source Code folder.

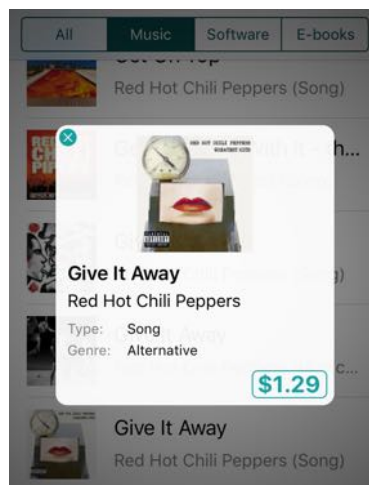
Note: Even though URLSession is pretty easy to use and quite capable, many developers prefer to use third-party networking libraries that are often even more convenient and powerful.

The most popular library at this point is AFNetworking, written in Objective-C but very usable from Swift (github.com/AFNetworking). A native Swift library is Alamofire (github.com/Alamofire).

I suggest you check out these libraries and see how you like them. Networking is such an important feature of mobile apps that it's worth being familiar with the different possible approaches to send data up and down the 'net.

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.

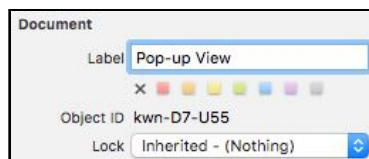
➤ Open the storyboard and drag a new **View Controller** into the canvas. Change its **Class** to **DetailViewController**.

➤ 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.

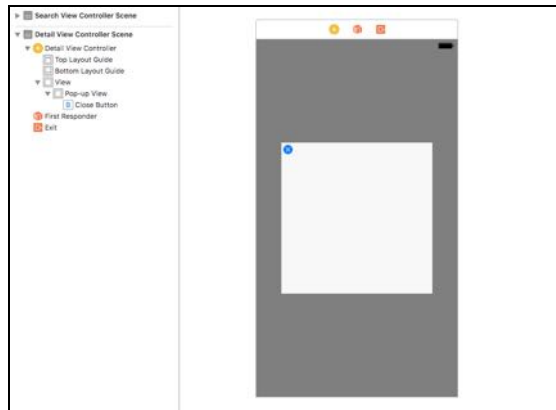


Giving the view a description for use in Xcode

➤ 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:

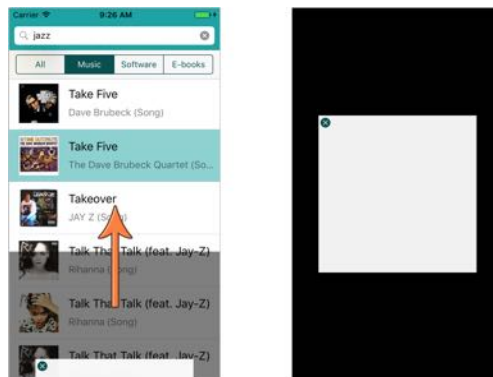
```
func tableView(_ tableView: UITableView,
               didSelectRowAt indexPath: IndexPath) {
    tableView.deselectRow(at: indexPath, animated: true)
    performSegue(withIdentifier: "ShowDetail", sender: indexPath)
}
```

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.

Now you need to tell the app that 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)  
        }  
    }  
}
```

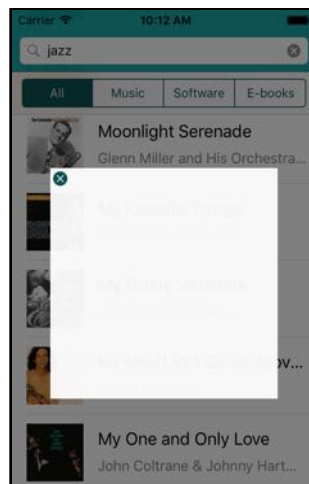
The methods from this delegate protocol tell UIKit what objects it should use to perform the transition to the Detail View Controller. It will now use your new `DimmingPresentationController` class instead of the standard presentation controller.

► 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.

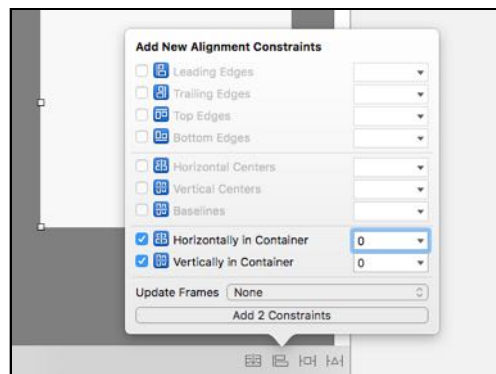
➤ Also verify that the close button works to dismiss the pop-up.

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? ■

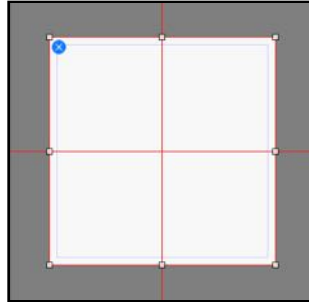
Answer: Add some Auto Layout constraints, of course! The current design of the Detail screen is for the iPhone SE. When the app runs on a larger device, UIKit doesn't know yet that it should keep the pop-up view centered.

➤ 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

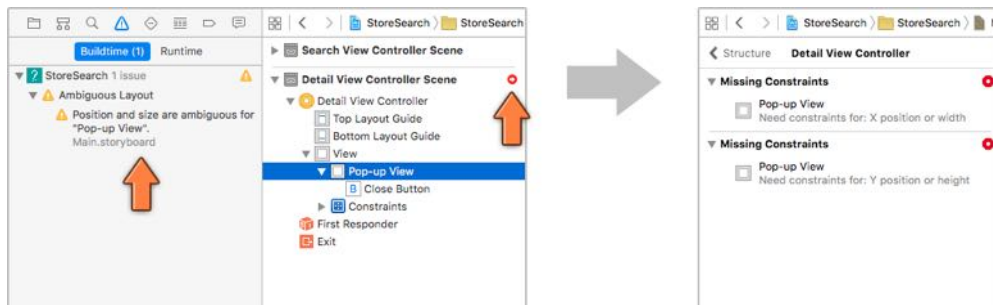
One small hiccup: these lines are supposed to be blue, not red. Whenever you see red or orange lines, Auto Layout has a problem.

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:

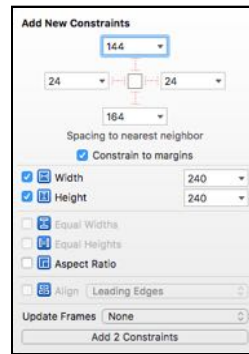


Xcode shows Auto Layout errors in the Issue navigator

➤ 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.

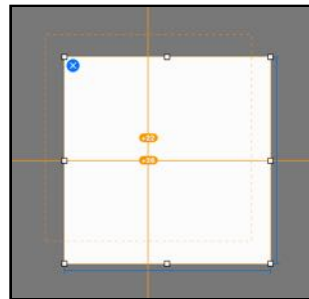
➤ Select the Pop-up View and click the **Pin** button. Put checkmarks in front of **Width** and **Height**. Click **Add 2 Constraints** to finish.



Pinning the width and height of the Pop-up View

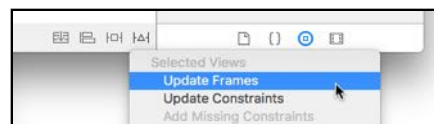
Now the lines turn blue and Auto Layout is happy.

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 **Resolve Auto Layout Issues** button at the bottom (to the right of the Pin button) and choose **Update Frames**:

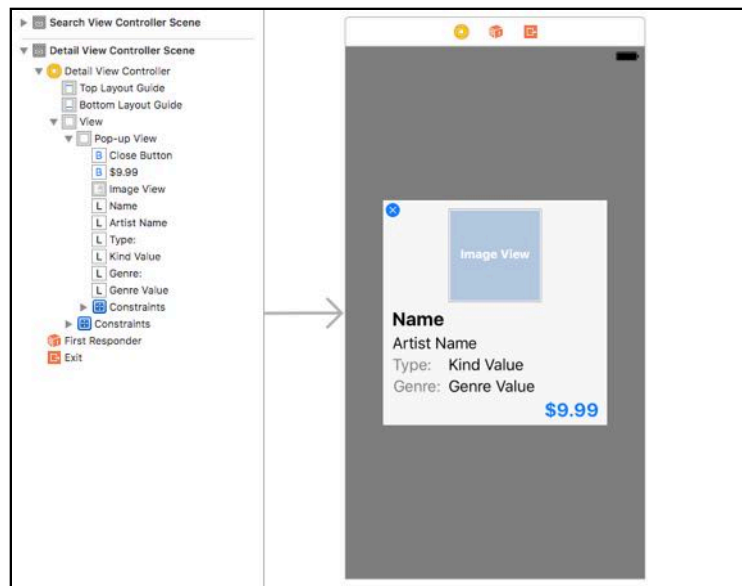


The Resolve Auto Layout Issues menu

► 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.

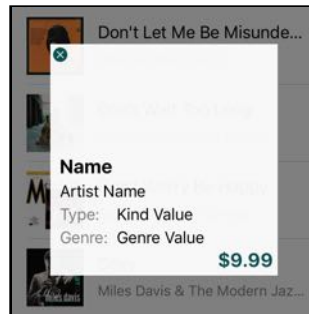
► You shouldn't have to change the font for the other labels; they use the default value of System 17.

- 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**:

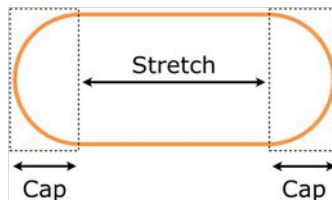
```
@IBOutlet weak var popupView: UIView!  
@IBOutlet weak var artworkImageView: UIImageView!  
@IBOutlet weak var nameLabel: UILabel!  
@IBOutlet weak var artistNameLabel: UILabel!  
@IBOutlet weak var kindLabel: UILabel!  
@IBOutlet weak var genreLabel: UILabel!  
@IBOutlet weak var priceButton: UIButton!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

- 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

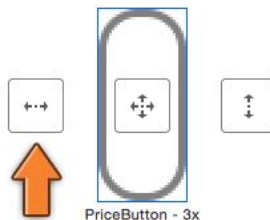
If you take a detailed look at this image you will see that it is only 11 points wide. That means it has a 5-point cap on the left, a 5-point cap on the right, and a 1-point body that will be stretched out.

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:



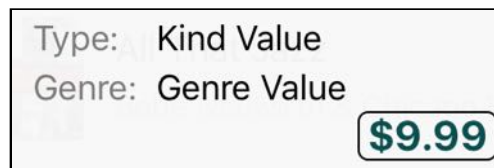
After slicing

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.

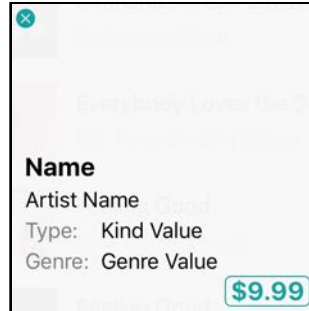
I like the dark green tint color in the rest of the app but for this pop-up it's a bit too dark. You can change the tint color on a per-view basis; if that view has subviews the new tint color also applies to these subviews.

► 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.

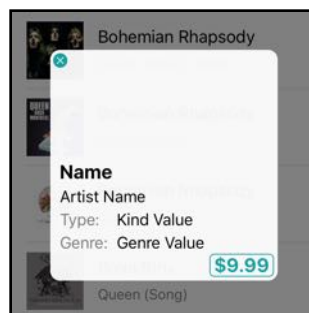
`UIView`s do their drawing using a so-called `CALayer` object. The `CA` prefix stands for Core Animation, which is the awesome framework that makes animations so easy on the iPhone. You don't need to know much about those "layers", except that each view has one, and that layers have some handy properties.

➤ 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.)

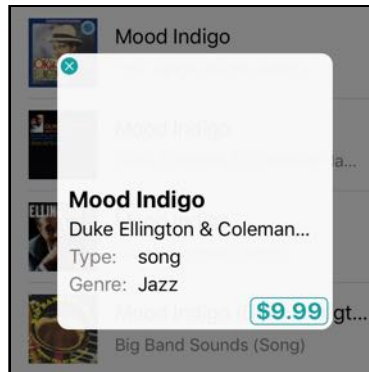
The Detail pop-up is launched with a segue triggered from `SearchViewController`'s `tableView didSelectRowAt`. You'll have to add a `prepare(for:sender:)` method to configure the `DetailViewController` when the segue happens.

► 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.

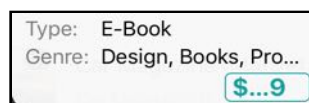
There are different rules for displaying various currencies, especially if you take the user's language and country settings into consideration. You could program all of these rules yourself, which is a lot of effort, or choose to ignore them. Fortunately, you don't have to make that tradeoff because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

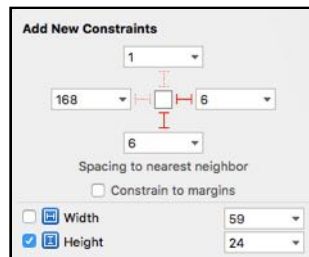
When you designed the storyboard you made this button 68 points wide. You didn't put any constraints on it, so Xcode gave it an automatic constraint that always forces the button to be 68 points wide, no more, no less.

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 **Pin** 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

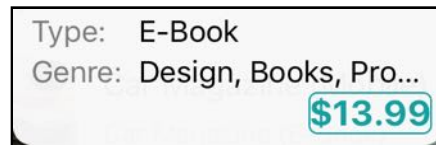
The orange lines simply mean that the current position and/or size of the button in the storyboard does not correspond to the position and size that Auto Layout calculated from the constraints. This is easily fixed:

► 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.

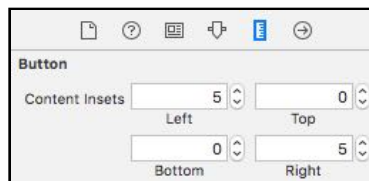
- Pinned to the right edge of the pop-up with a distance of 6 points. When the button needs to grow to accommodate larger prices, it will extend towards the left. Its right edge always stays aligned with the right edge of the pop-up.
 - Pinned to the bottom of the pop-up, also with a distance of 6 points.
 - 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.

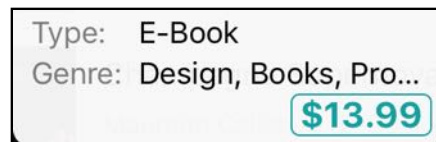
- Go to the **Size inspector** and find where it says **Content Insets**. Change **Left** and **Right** to 5.



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.

- Choose **Update Frames** from the **Resolve Auto Layout Issues** menu 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:

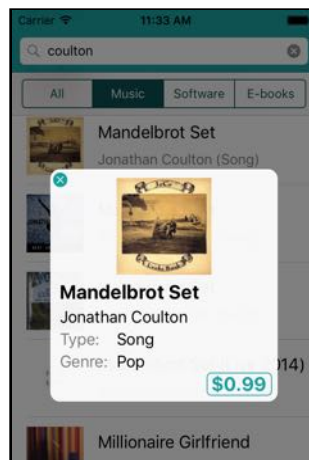
```
deinit {  
    print("deinit \(self)")  
    downloadTask?.cancel()  
}
```

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.)

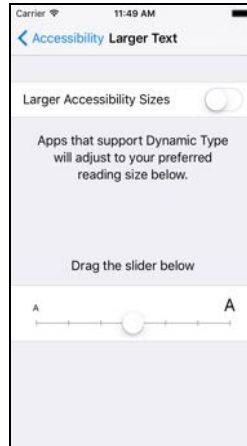
► This is a good time to commit the changes.

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:

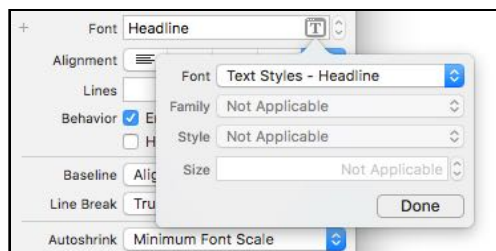


The Larger Text accessibility settings

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.

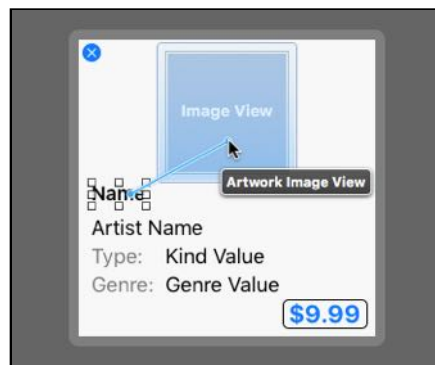
You want to make the name label resizable so that it can hold any amount of text at any possible font size, but it cannot go outside the bounds of the pop-up, nor overlap the labels below.

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

Previously you've used the Pin 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?

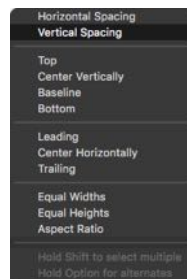
If you use the Pin button on the Name label, Interface Builder may decide to pin it to the bottom of the close button, which is weird. It makes more sense to pin the Name label to the image view instead. That's why you're going to use a different way to make constraints.

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



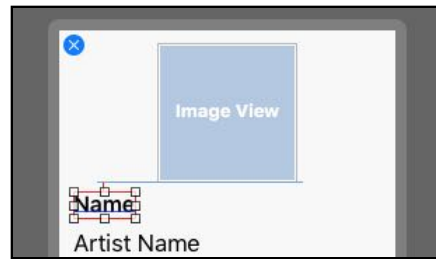
Ctrl-drag to make a new constraint between two views

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:



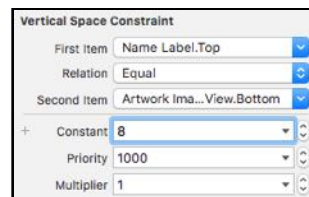
The new vertical space constraint

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.

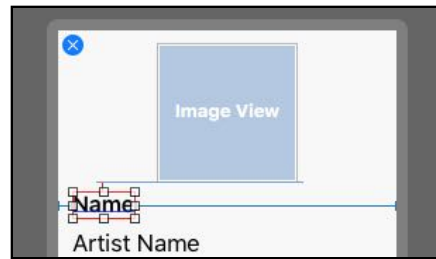
► Select the **Name** label again and **Ctrl-drag** to the left and connect it to **Pop-up View**. From the pop-up choose **Leading Space to Container**:



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**.



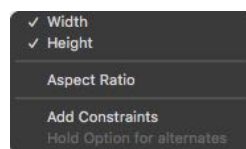
The constraints for the Name label

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.)

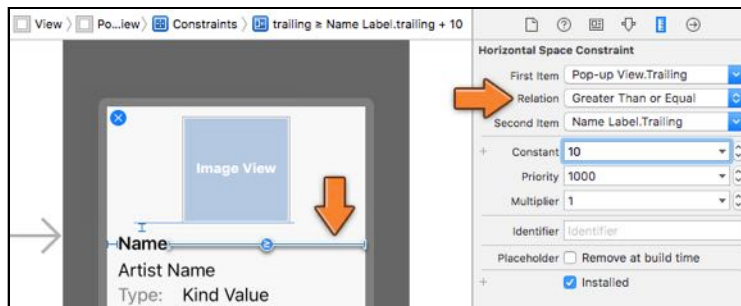


Adding multiple constraints at once

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 choose Update Frames from the Resolve Auto Layout Issues menu.

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:



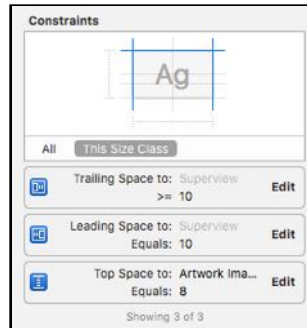
The text overlaps the other labels

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, choose **Resolve Auto Layout Issues** → **Update Frames**. This fixes any orange thingies.

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

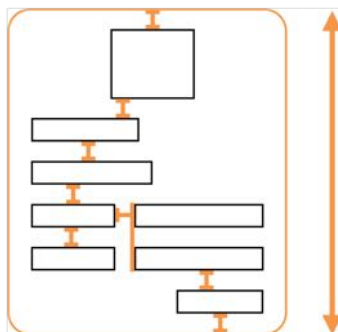
- Pin it to the left with a Leading Space to Container.
- Pin it to the Type: label with a Vertical Spacing, size 4.

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? ■

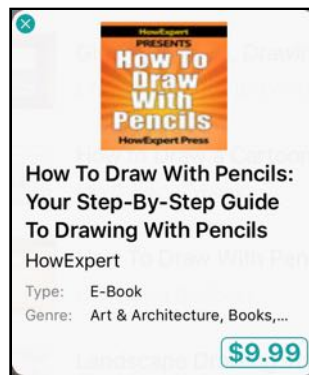
Answer: The Pop-up View still has a Height constraint that forces it to be 240 points high. But the labels and the vertical space constraints don't add up to 240.

➤ 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.

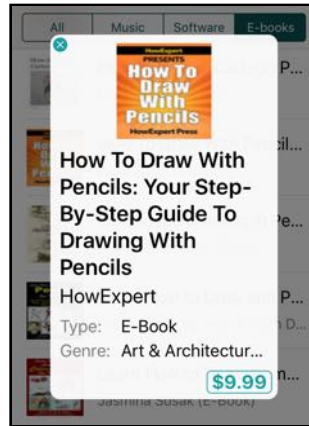


The text properly wraps without overlapping

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 a 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). 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

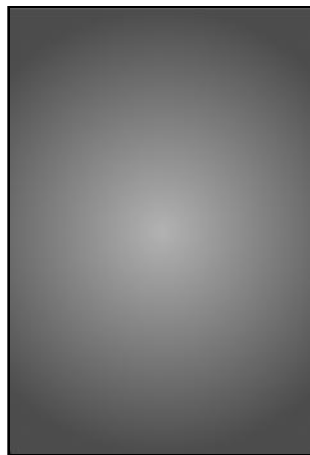
As you can see in the previous screenshots, the table view in the background is dimmed by the view of the DetailViewController, which is 50% transparent black. That allows the pop-up to stand out more.

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`,

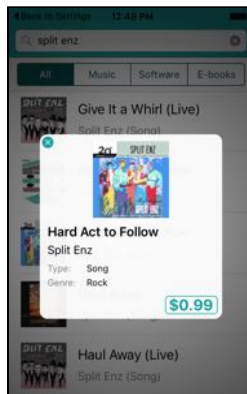
and it contains the views from the DetailViewController. So this piece of logic places the GradientView in between those two screens.

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!

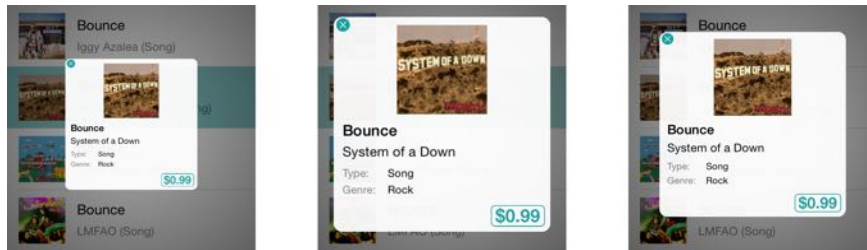
To make this animation happen you have to tell the app to use the new animation controller when presenting the Detail pop-up. That happens in the transitioning delegate inside **DetailViewController.swift**.

► 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()  
}
```

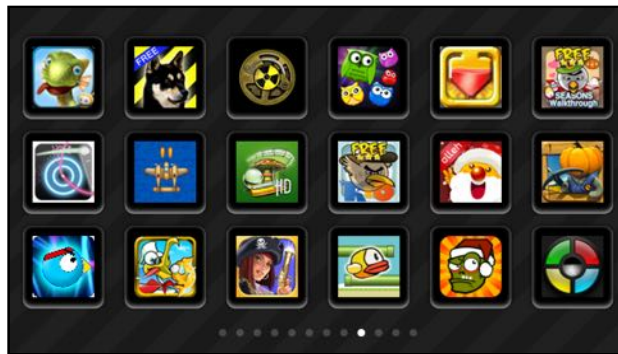
- Run the app and try it out. That looks pretty sweet if you ask me!
- If you're happy with the way the animation looks, then commit your changes.

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:

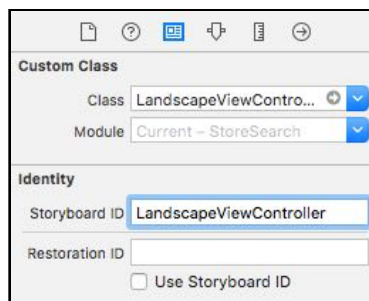
- Create a new view controller and show that when the device is rotated. Hide this view controller when the device returns to the portrait orientation.
- Put some fake buttons in a UIScrollView, in order to learn how to use scroll views.
- Add the paging control (the dots at the bottom) so you can page 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 landscape.



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

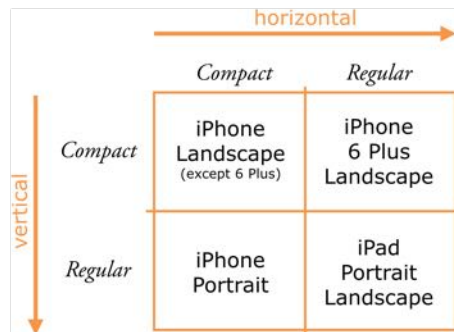
Whenever one or more of these traits change, for whatever reason, UIKit calls

`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 may be 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:
                    UINavigationControllerTransitionCoordinator) {
    // 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.

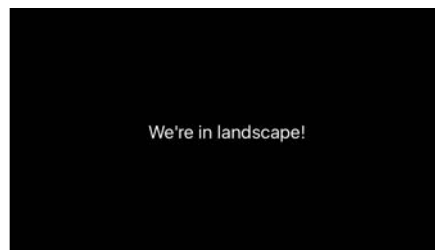
One of the reasons you’re not using a modal segue for the Landscape screen in this app, even though it is a full-screen view controller, is that the Detail pop-up already is modally presented and this could potentially cause conflicts. Besides, I wanted to show you a fun alternative to modal segues.

- To get the app to compile, add an empty implementation of the “hide” method:

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

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:
                    UINavigationControllerTransitionCoordinator) {
    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:
                    UINavigationControllerTransitionCoordinator) {
    . . .
    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 `UINavigationControllerTransitionCoordinator` 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)`:

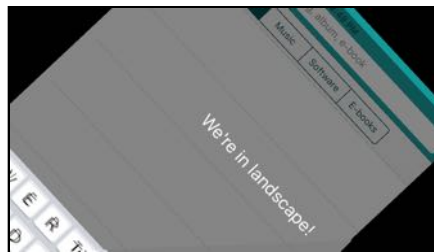
```
func hideLandscape(with coordinator:
                   UINavigationControllerTransitionCoordinator) {
    if let controller = landscapeViewController {
```

```
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:
                    UINavigationControllerTransitionCoordinator) {
    . . .
    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 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.

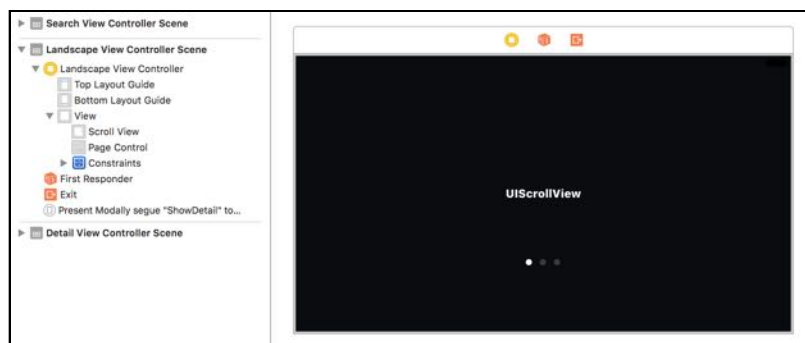
Important: Do not place the Page Control *inside* the Scroll View. They should be at the same level in the view hierarchy:



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!
```

Next up you'll disable Auto Layout for this view controller. The storyboard has a "Use Auto Layout" checkbox but you cannot use that. It would turn off Auto Layout for all the 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 frames 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 frames 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 !.

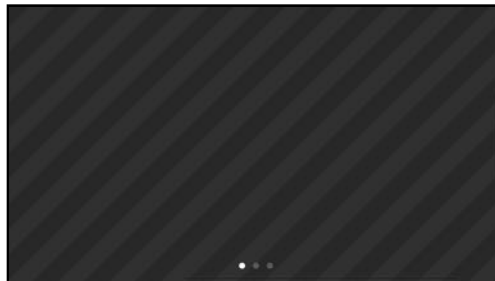
➤ Also add the following line to viewDidLoad():

```
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:



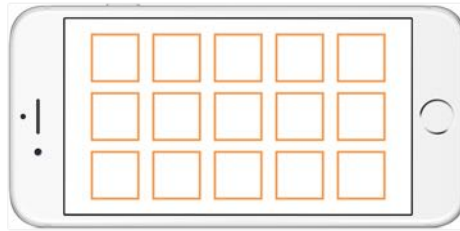
The scroll view now has a background image and it can scroll

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 $3 \times 5 = 15$ 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 $568 - 94 \times 6 = 4$ 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:
                    UINavigationControllerTransitionCoordinator) {
    . . .
    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.

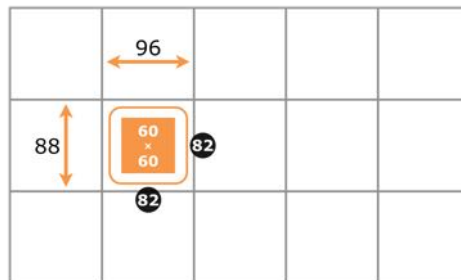
Note: Wouldn't it be possible to come up with a nice formula that calculates all this stuff for you, rather than *hard-coding* these sizes and margin values? Probably, but it won't be easy. There are two things you want to optimize for: getting the maximum number of rows and columns on the screen, but 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.

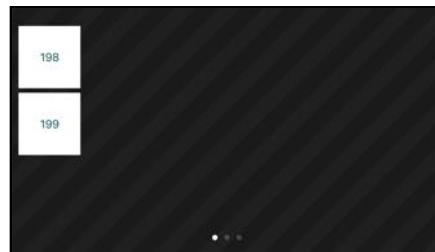
I also threw in a `print()` for good measure, so you can verify that you really end up with the right amount of pages.

► 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 ($\text{width}/2$), 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.

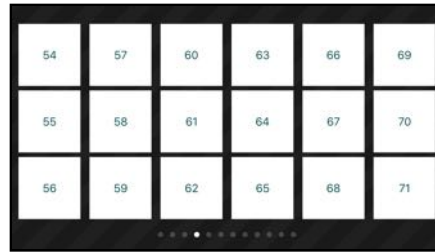
The transition from one page to another after tapping in the page control is still a little abrupt, though. An animation would help here.

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

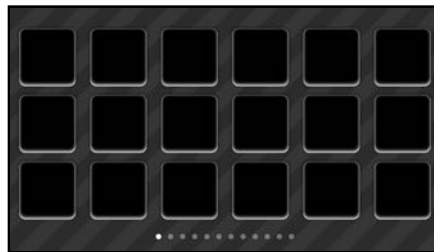
First let's give the buttons a nicer look.

- 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 line to `tileButtons()` to call this new method, right after where 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 <code>SearchResult</code> 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 state 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

            self.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 {
                    self.state = .noResults
                } else {
                    searchResults.sort(by: <)
                    self.state = .results(searchResults)
                }
                success = true
            }

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

Instead of the old variables `isLoading`, `hasSearched`, and `searchResults`, this now

only changes state.

There is a lot that can go wrong between performing the network request and parsing the JSON. By setting `self.state` 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 is 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, `state` becomes `.noResults`.

The interesting thing happens when the array is *not* empty. After sorting it like before, you do `self.state = .results(searchResults)`. This gives `state` 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 `state`.

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.

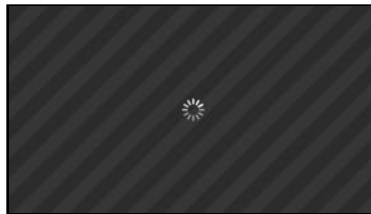
► Also add the new `showSpinner()` method:

```
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
```

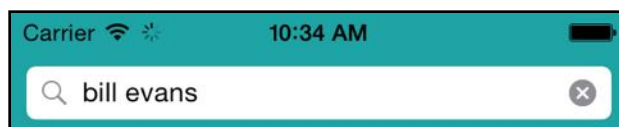
► Add the following line to `performSearch(for, category, completion)`, just before starting the search:

```
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 {
    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:

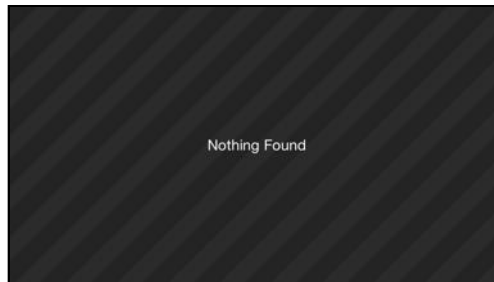
```
width = ceil(width/2) * 2
```

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()`.

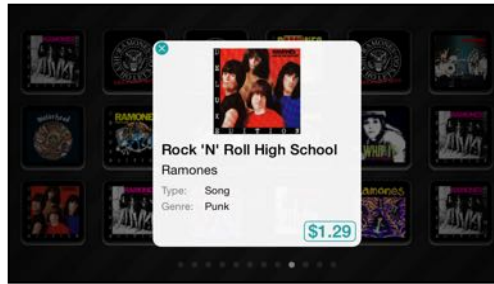
► Change the switch statement in that method to:

```
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
        }
    }
}

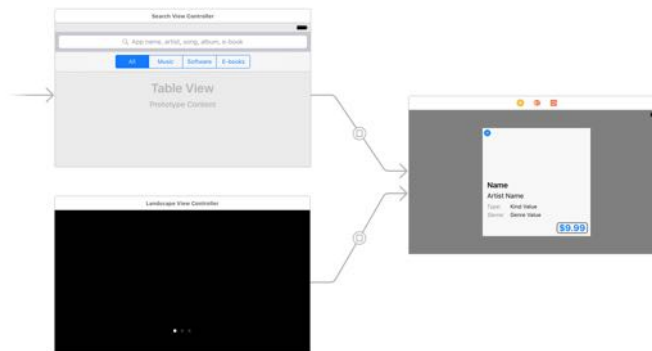
```

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.

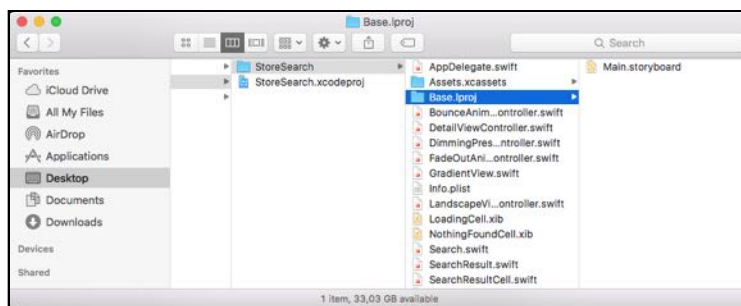
Internationalization

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

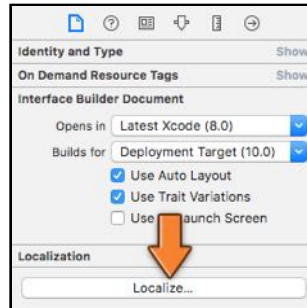
There is a subfolder named **Base.lproj** that contains one file, **Main.storyboard**. The Base.lproj folder is for files that can be localized. So far that's only the storyboard but you'll add more files to this folder soon.

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).

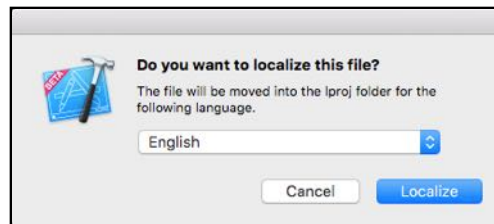
Because the NothingFoundCell.xib file isn't in any XX.lproj folders, it does not have any localizations yet.



The NothingFoundCell has no localizations

► Click the **Localize...** button in the Localization section.

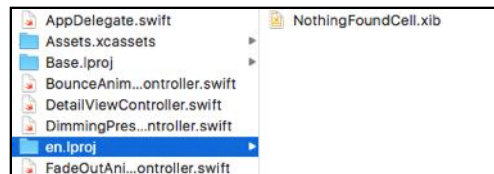
Xcode asks for confirmation because this involves moving the file to a new folder:



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

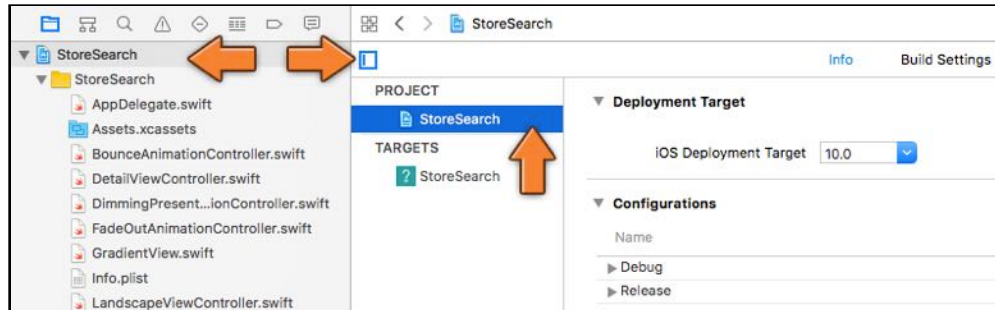
The **File inspector** for **NothingFoundCell.xib** now lists English as one of the localizations.



The Localization section now contains an entry for English

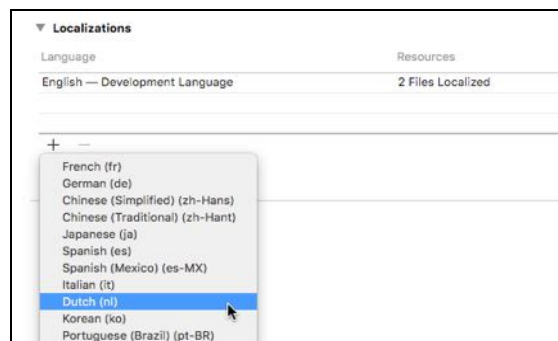
To add a new language you have to switch to the **Project Settings** screen.

- Click on **StoreSearch** at the top of the Project navigator to open the settings page.



The Project Settings

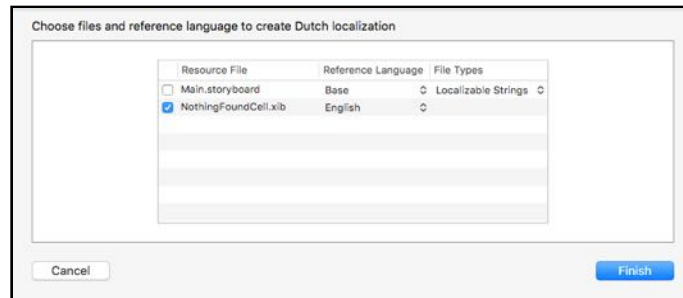
- From the sidebar, choose **StoreSearch** under **PROJECT** (not under TARGETS). If the sidebar isn't visible click the small icon at the top to open it.
- In the **Info** tab, under the **Localizations** section press the **+** button:



Adding a new language

- From the pop-up menu choose **Dutch (nl)**.

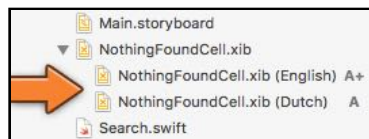
Xcode now asks which resources you want to localize. Uncheck everything except for **NothingFoundCell.xib** and click **Finish**.



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 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.

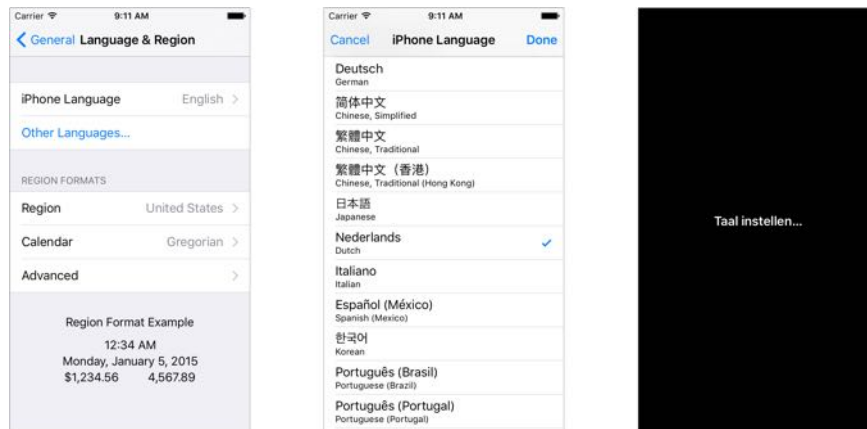
If you run the app now, nothing will have changed. You have to switch the Simulator to use the Dutch language first. However, before you do that you really should remove the app from the simulator, clean the project, and do a fresh build.

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.

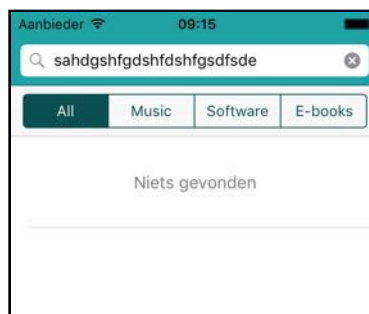
- Remove the app from the Simulator. Do a clean (**Product** → **Clean** or **Shift-⌘-K**) and re-build the app.
- Open the **Settings** app in the Simulator and go to **General** → **Language & Region** → **iPhone Language**. From the list pick **Nederlands (Dutch)**.



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**.)

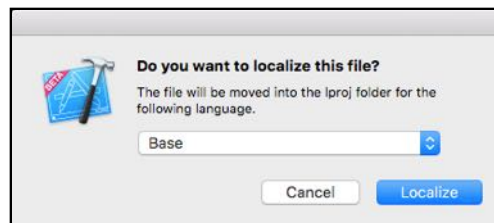
To localize the other nibs you could repeat the process and add copies of their xib files to the **nl.lproj** folder. That isn't too bad for this app but if you have an app with really complicated screens then having multiple copies of the same nib can become a maintenance nightmare.

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**:

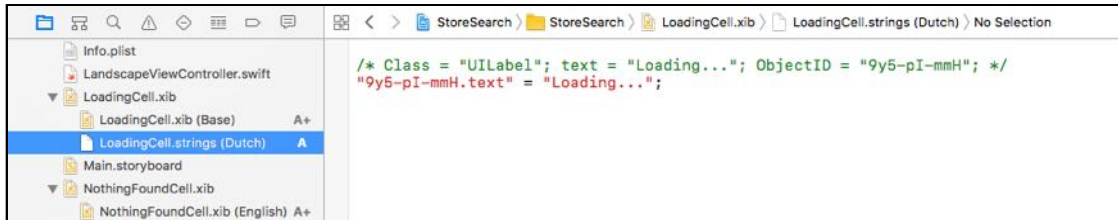


Adding a Dutch localization

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:

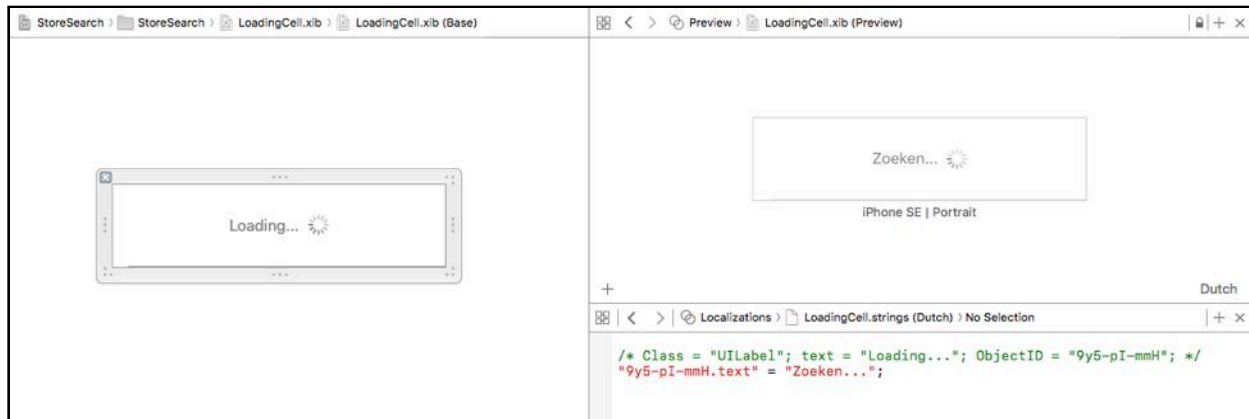
```
/* Class = "UILabel"; text = "Loading..."; ObjectID = "9y5-pI-mmH"; */  
"9y5-pI-mmH.text" = "Loading...";
```

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...**

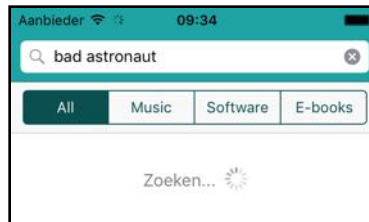
Tip: You can use the Assistant editor in Interface Builder to get a preview of your localized nib. Go to **LoadingCell.xib (Base)** and open the Assistant editor. From the Jump bar at the top, choose **Preview**. In the bottom-right corner it says English. Click this to switch to a Dutch preview.



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**.

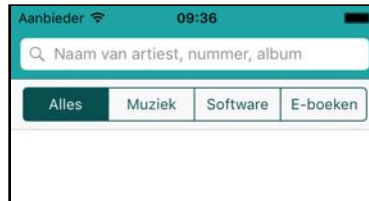
```
"68e-CH-NSs.placeholder" = "Naam van artiest, nummer, album";
```

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:**

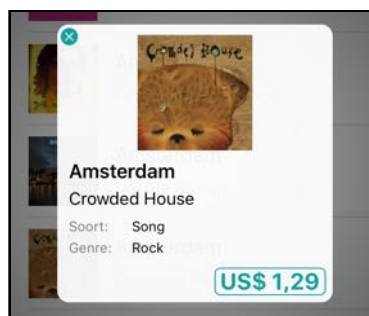
```
"DCQ-US-EVg.text" = "Soort:";
```

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";
"x0H-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.

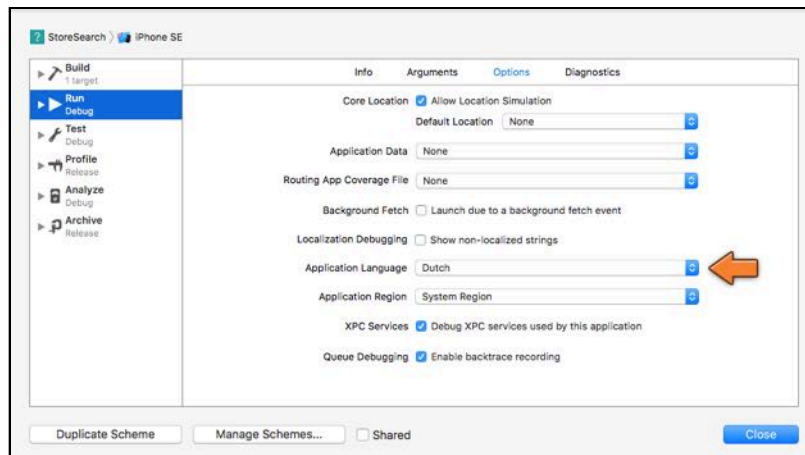
The Landscape View Controller doesn't have any text to translate.

► 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.

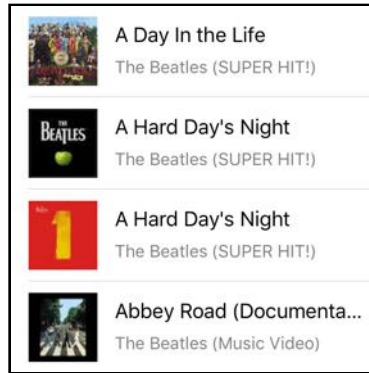
Tip: It's a good idea to make these comments as detailed as you can. In the words of fellow tutorial author Scott Gardner:

"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 a 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:

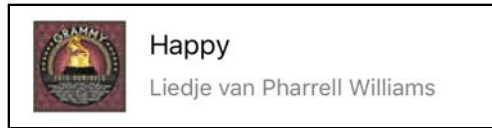
```
/* Format for artist name label */
"%@ (%@)" = "%1$@ (%2$@)";
```

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 displayNameForKind = [
    "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 displayNameForKind[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 = displayNameForKind[kind] {
    return name
} else {
    return kind
}

```

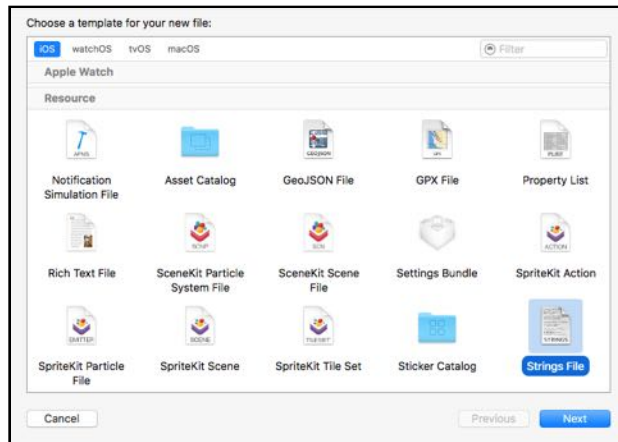
but shorter!

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

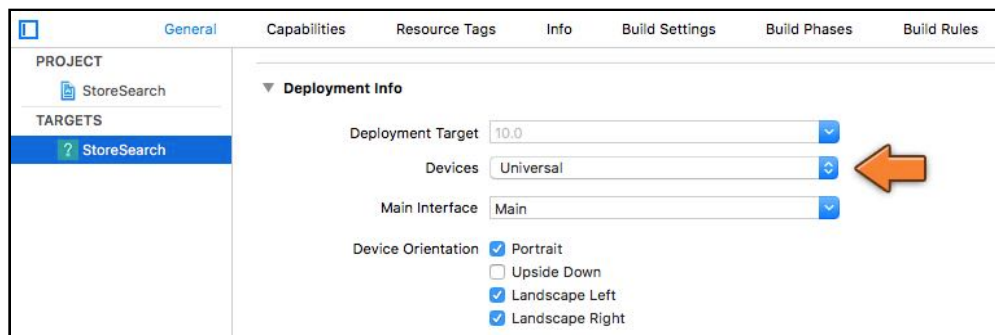
Even though the apps you've written so far are only for the iPhone, everything you have learned also applies to writing iPad apps.

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.

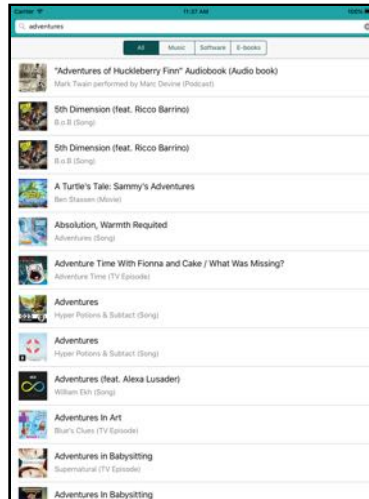
In the **General** tab under **Deployment Info** there is a setting for **Devices**. It is currently set to iPhone, but change it to **Universal**.



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

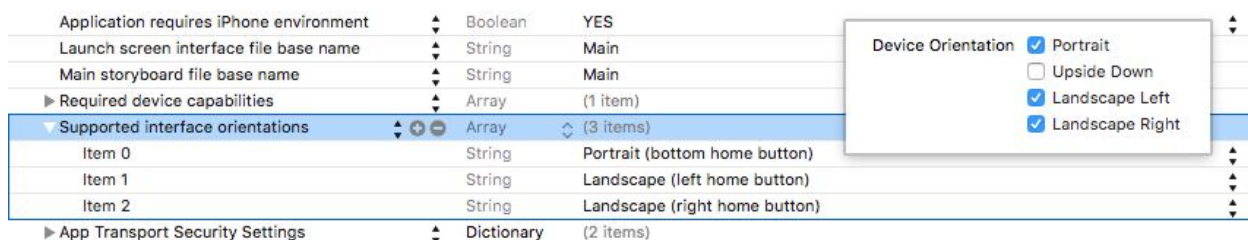
If the iPad is in landscape, the split-view controller has enough room to show both panes at the same time. However, in portrait mode only the detail view controller is visible and the app provides a button that will slide the master pane into view. (You can also swipe the screen to reveal and hide it.)

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:

| | | | | |
|---|---|--------|-------------------------------|----|
| ▼ Supported interface orientations | ↕ | Array | (3 items) | |
| Item 0 | | String | Portrait (bottom home button) | ▲▼ |
| Item 1 | | String | Landscape (left home button) | ▲▼ |
| Item 2 | | String | Landscape (right home button) | ▲▼ |
| ▼ Supported interface orientations (iPad) | ↕ | Array | (4 items) | |
| Item 0 | | String | Portrait (bottom home button) | ▲▼ |
| Item 1 | | String | Portrait (top home button) | ▲▼ |
| Item 2 | | String | Landscape (left home button) | ▲▼ |
| Item 3 | | String | Landscape (right home button) | ▲▼ |

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.

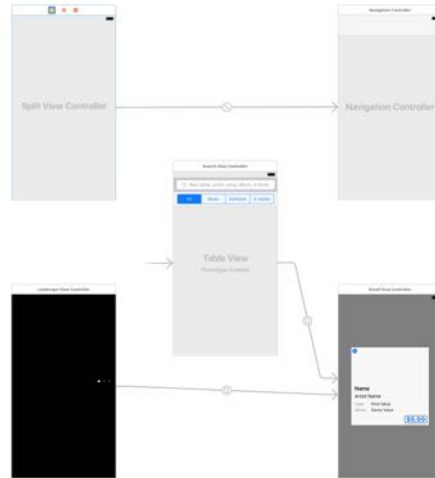
Thanks to universal storyboards you can simply add a Split View Controller object to the storyboard. The split-view is only visible on the iPad; on the iPhone it stays hidden.

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?

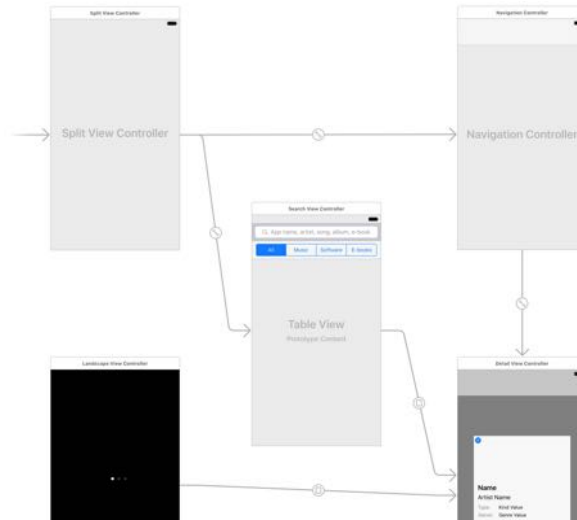
➤ Ctrl-drag from the Split View Controller to the Navigation Controller. Choose **Relationship Segue – detail view controller**.

➤ 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.

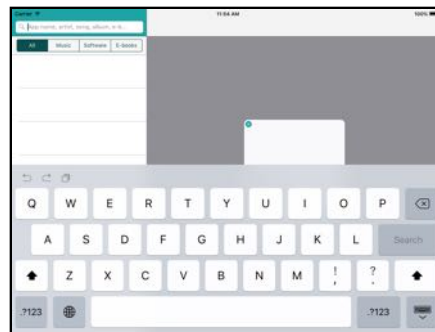
➤ Pick up the arrow that points to Search View Controller and drag it onto the Split View Controller. (You can also check the **Is Initial View Controller** option in the Attributes inspector.)

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

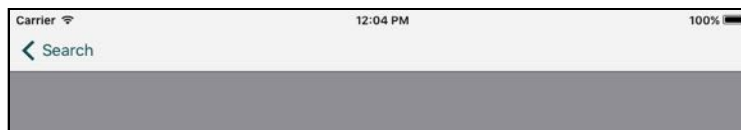
It would be better if this back button said “Search”. You can fix that by giving the view controller from the master pane a title.

► 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**:

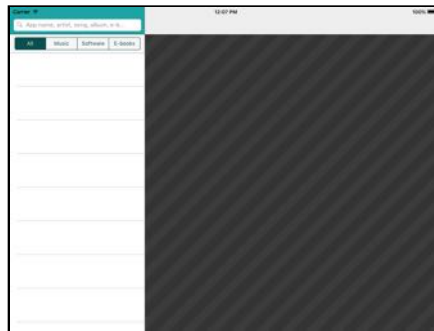
```
var isPopUp = false
```

► 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.

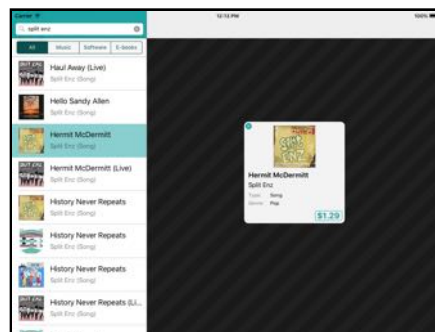
In that case you don't want to call `updateUI()` as there is no user interface yet to update. The `isViewLoaded` check ensures this property observer only gets used when on an iPad.

► 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`.

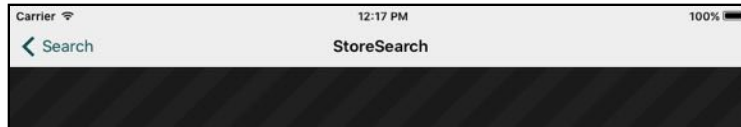
Because `NSBundle`'s `localizedInfoDictionary` can be `nil` you need to unwrap it. The value stored under the `"CFBundleDisplayName"` key may also be `nil`. And finally, the `as?` cast to turn the value to a `String` can also potentially fail. If you're counting along, that is three things that can go 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.

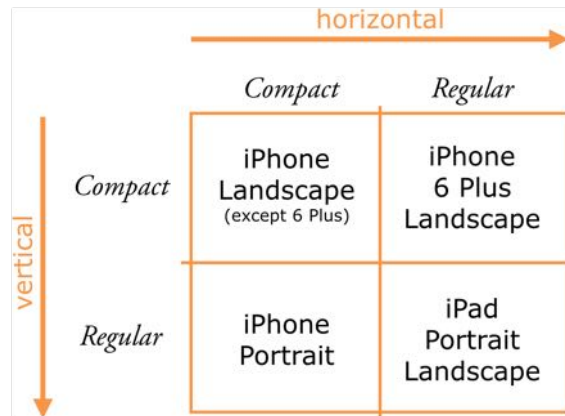


Size classes in the View as: pane

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.

☐ View as: iPad Pro 9.7" (wR hR)

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

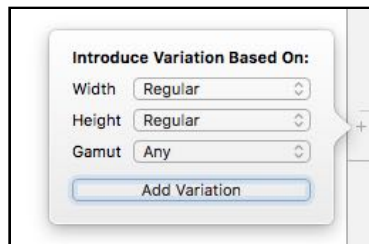
the bottom, to the **Installed** option.



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:

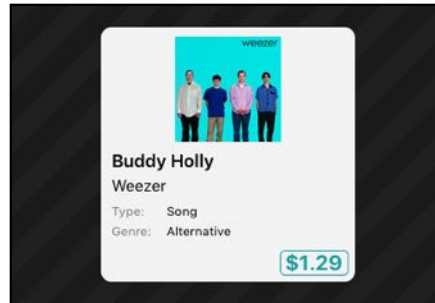


The Close Button is still present but grayed out

➤ 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:



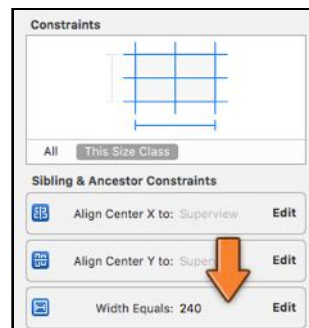
No more close button in the top-left corner

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.

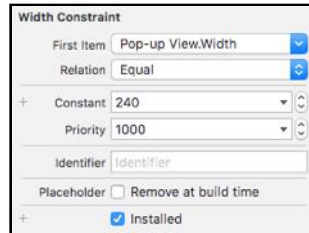
➤ Select the **Pop-up View** and go to the **Size inspector**. The **Constraints** section shows the constraints for this view:



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.

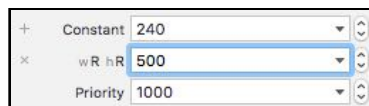
➤ Double-click **Width Equals: 240**. This brings up the **Size inspector** for just that constraint:



The Size inspector for the Width constraint

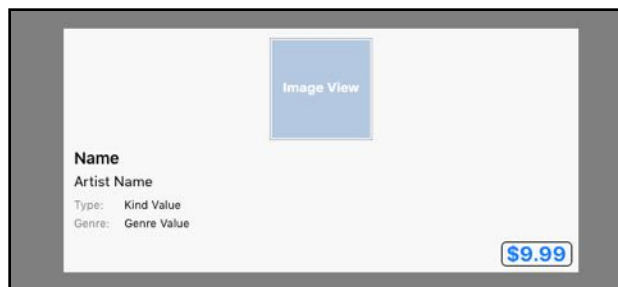
(If you just type in a new value for Constant, the constraint will become larger for all size classes again.)

➤ 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**.

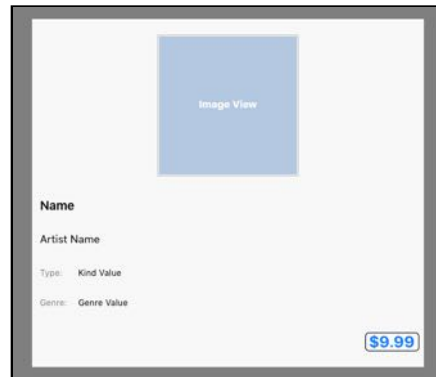
➤ Select the **Vertical Space** constraint between the **Name** label and the **Image View** and go to its **Size inspector**. Add a new variation for Constant and type **28** into the **wR hR** field.

➤ 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:

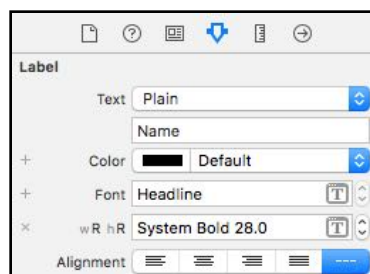


The Pop-up View after changing the vertical spaces

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

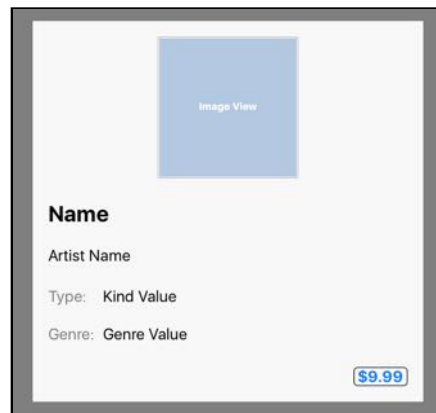
If orange lines appear, use the Resolve Auto Layout Issues menu to update the positions and sizes of the views so they correspond with the constraints again (tip: choose Update Frames from the "All Views" section).

- 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.

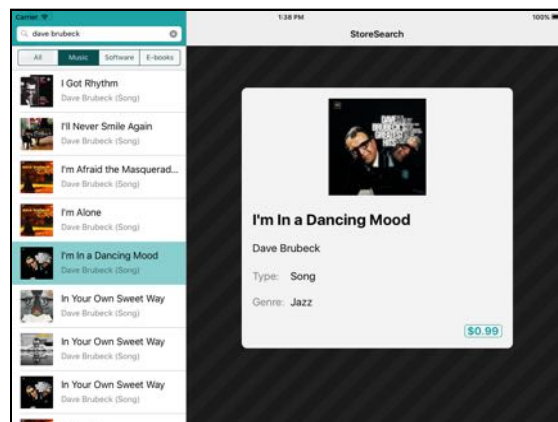
The final layout should look like this:



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.

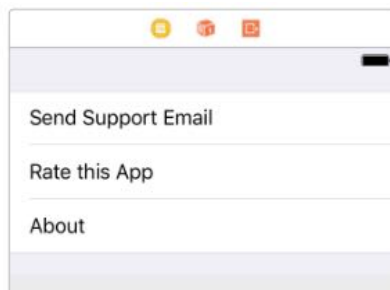
The **View as:** panel has a button **Vary for Traits**. You can use this to change how a view controller acts when it is part of such a split screen.

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**):



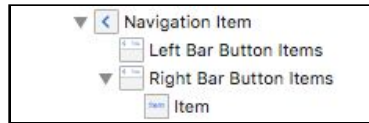
The design for the new table view controller

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.

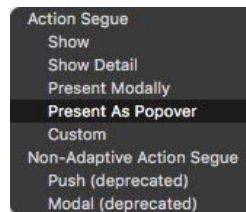


The new bar button item in the Navigation Item

- 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.

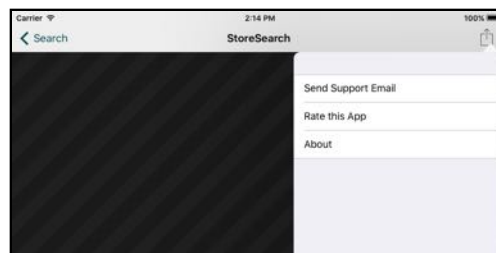
- Ctrl-drag from the bar button (in the outline pane) to the Table View Controller to make segue. Choose segue type **Action Segue – Present As Popover**.



The new bar button item in the Navigation Item

- Give the segue the identifier **ShowMenu**.

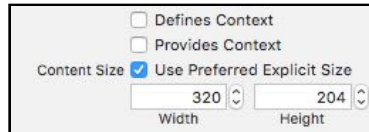
If you run the app and press the menu button, the app looks like this:



That menu is a bit too tall

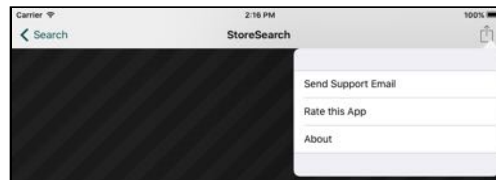
The popover doesn't really know how big its content view controller is, so it just picks a size. That's ugly, but you can tell it how big the view controller should be with the *preferred content size* property.

- 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

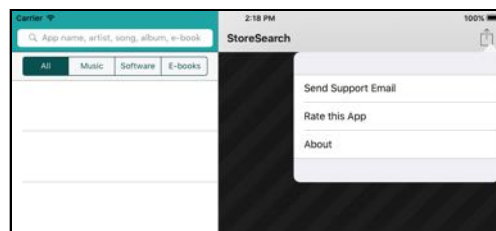
Now the size of the menu popover looks a lot more appropriate:



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()`:

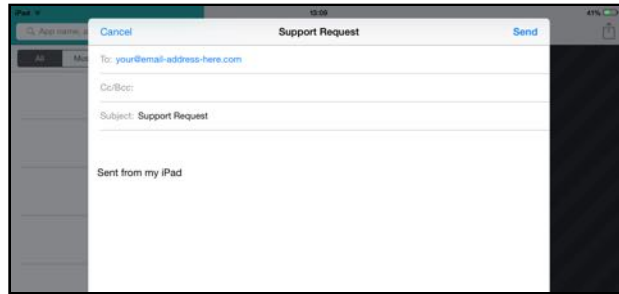
```
dismiss(animated: true) {  
    if MFMailComposeViewController.canSendMail() {  
        let controller = MFMailComposeViewController()  
        controller.setSubject(NSLocalizedString("Support Request",  
                                                comment: "Email subject"))  
        controller.setToRecipients(["your@email-address-here.com"])  
        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**.

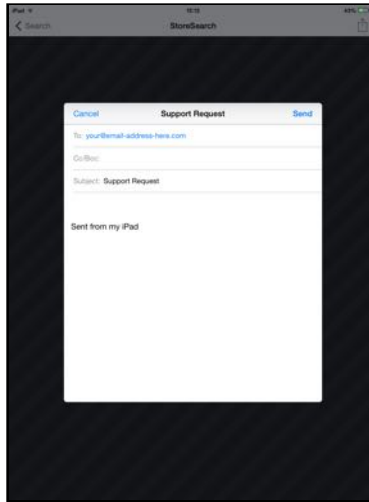
On the iPhone if you presented a modal view controller it always took over the entire screen, but on the iPad you have several options.

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

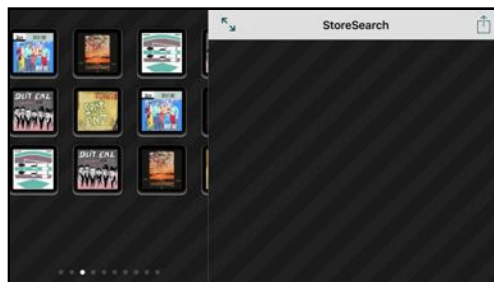
A form sheet is smaller than a page sheet so it always takes up less room than the entire screen. There is also a “full screen” presentation style that always covers the entire screen, even in landscape. Try it out!

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.

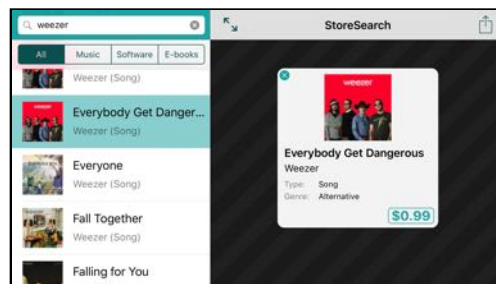
What about the size class? That sounds like it would be the obvious thing to tell the different devices apart. Unfortunately, looking at the size class *doesn't* work.

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:



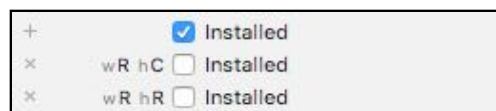
The app on the iPhone 7 Plus with a split-view

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:

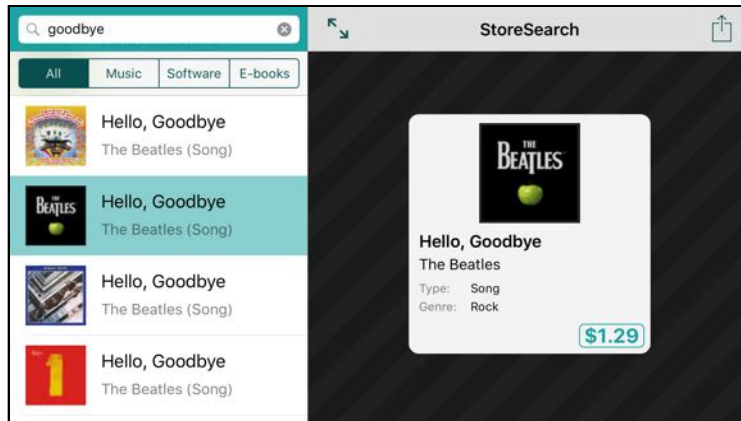


Adding a variation for size class width regular, height compact

► 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 6s 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.

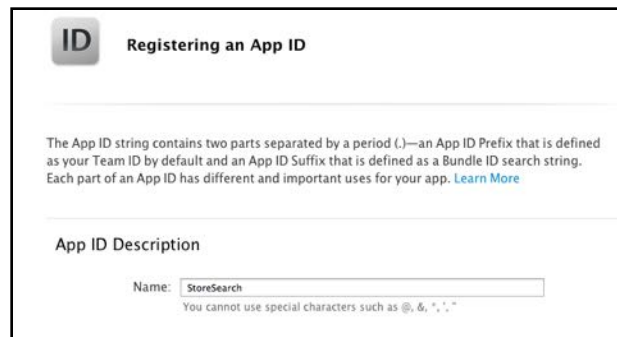
➤ Open your favorite web browser and surf to the Developer Member Center at <https://developer.apple.com/membercenter/>. Sign in and go to **Certificates**,

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.

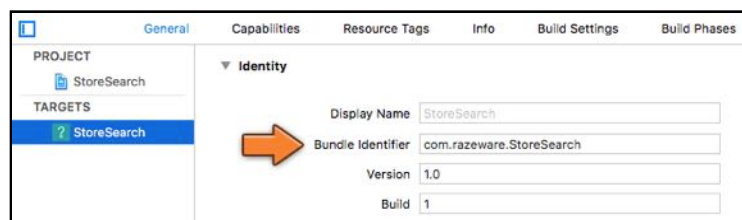
- Click on **App IDs** under **Identifiers** in the sidebar. In the new page that appears, press the **+** button to add a new App ID:



The screenshot shows the 'Registering an App ID' page. It has a title 'ID Registering an App ID'. Below the title, there is explanatory text: 'The App ID string contains two parts separated by a period (.)—an App ID Prefix that is defined as your Team ID by default and an App ID Suffix that is defined as a Bundle ID search string. Each part of an App ID has different and important uses for your app. [Learn More](#)'. Under the heading 'App ID Description', there is a 'Name:' label and a text input field containing 'StoreSearch'. Below the input field, a small note states: 'You cannot use special characters such as @, &, *, ', ;, ~'.

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 screenshot shows the Xcode interface with the 'Identity' tab selected. On the left sidebar, under 'PROJECT', 'StoreSearch' is listed. Under 'TARGETS', 'StoreSearch' is also listed and highlighted with a blue bar. An orange arrow points from the 'StoreSearch' target to the 'Bundle Identifier' field. The 'Bundle Identifier' field contains the text 'com.razeware.StoreSearch'. Other fields visible include 'Display Name' (StoreSearch), 'Version' (1.0), and 'Build' (1).

The Bundle ID must match with the identifier from Xcode

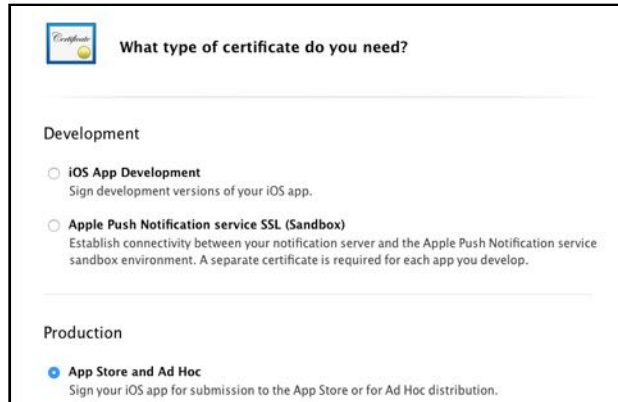
If you want your app to support push notifications, In-App Purchases, or iCloud then you can also configure that here. StoreSearch doesn't need any of that so leave the other fields on the default settings.

➤ 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.



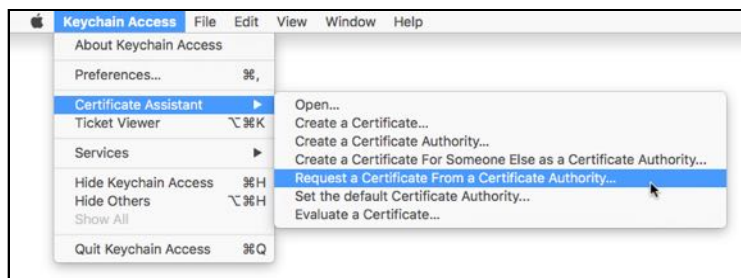
Creating a new distribution certificate

➤ 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:

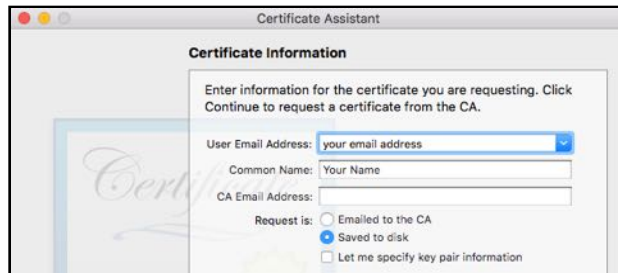
➤ Open the **Keychain Access** app on your Mac (it is in Applications/Utilities).

➤ 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.

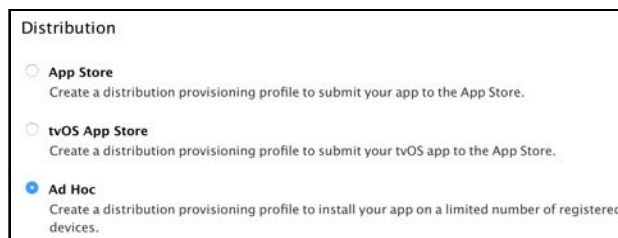
- Click the big **Download** button. This saves a file named **ios_distribution.cer** on your computer. Double-click this file to install it. You should be able to see the new certificate in the Keychain Access app under My Certificates.

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:

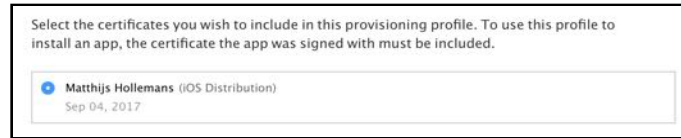


Creating a new provisioning profile for distribution

- 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:

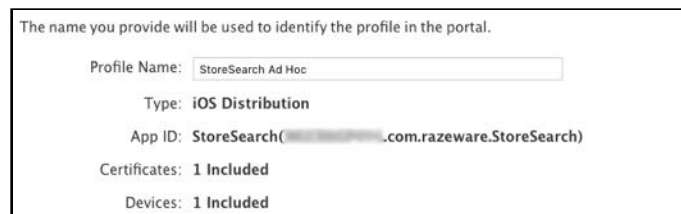


Selecting the certificate

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**.

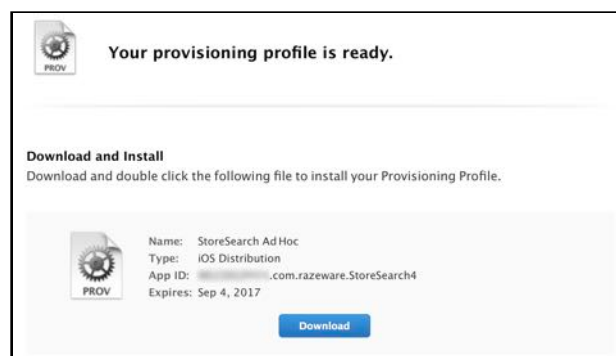
➤ Give the profile a name, for example **StoreSearch Ad Hoc**. Picking a good name is useful for when you have a lot of apps.



Giving the provisioning profile a name

➤ If everything looks OK, click **Generate**.

After a few seconds the provisioning profile is ready for download.



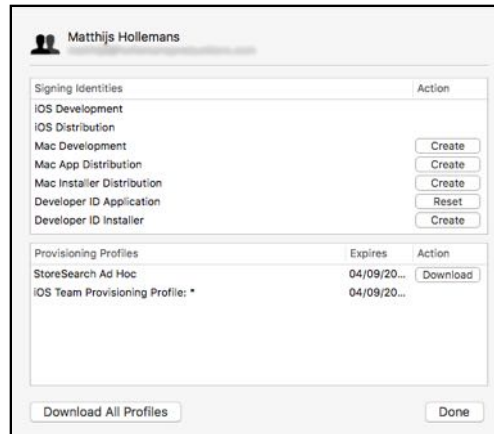
The provisioning profile was successfully created

➤ 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.

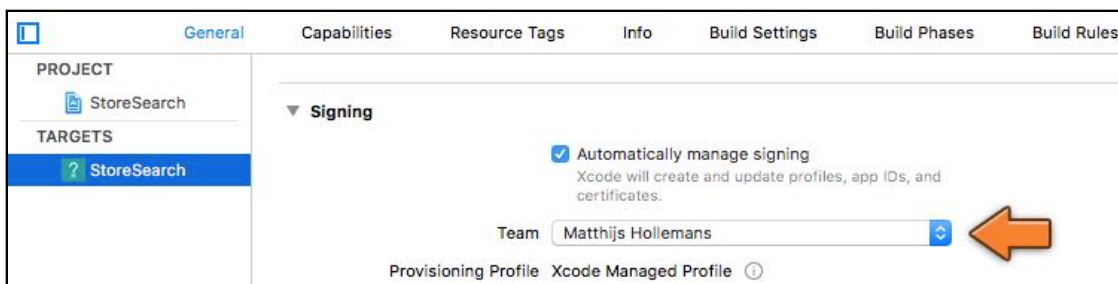
However, changing how your app gets built does mean that your app may act differently under certain circumstances. Therefore it's a good idea to give your app a thorough testing in Release mode as well, preferably by doing an Ad Hoc install on your own devices. That is the closest you will get to simulating 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.

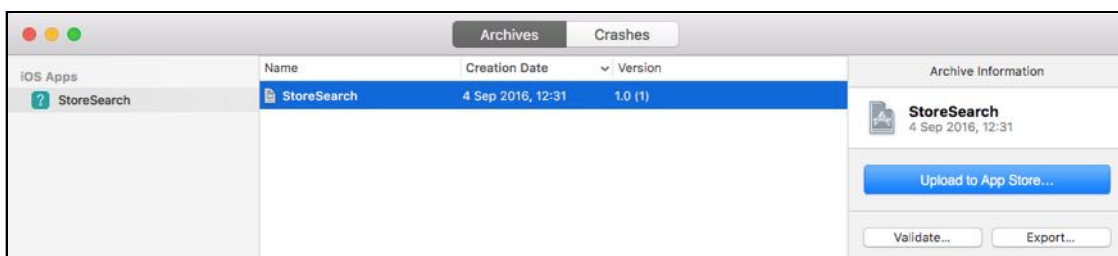


Choosing the team

- 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.

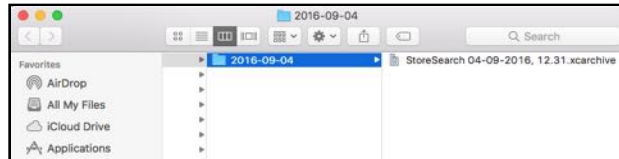
Now Xcode will build the app. By default, the Archive operation uses the Release build configuration.

- 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.

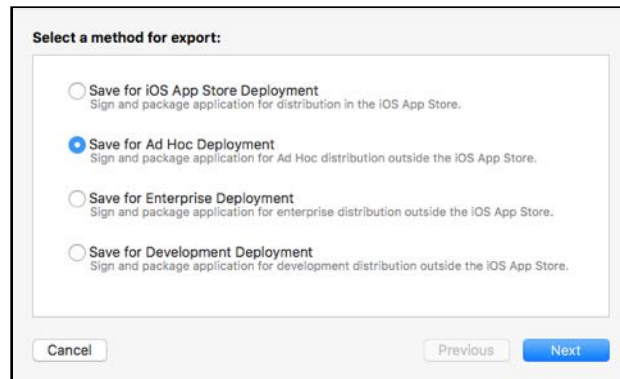
You don’t want to get crash reports that you can’t make any sense of! Even better, don’t make apps that crash, so you won’t get any crash reports at all...



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.



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/>

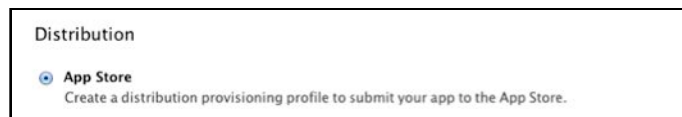


Submitting to the App Store

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.

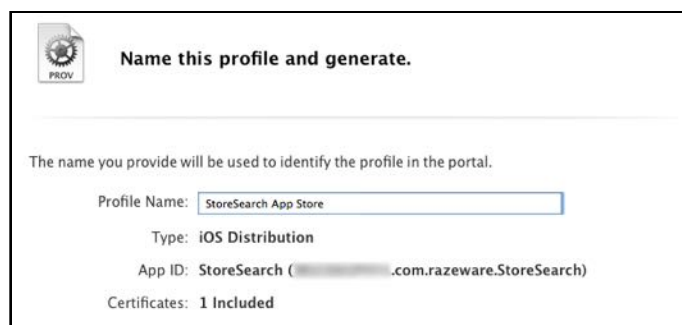
Doing so is actually fairly straightforward and I'll show you the steps here.

► 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

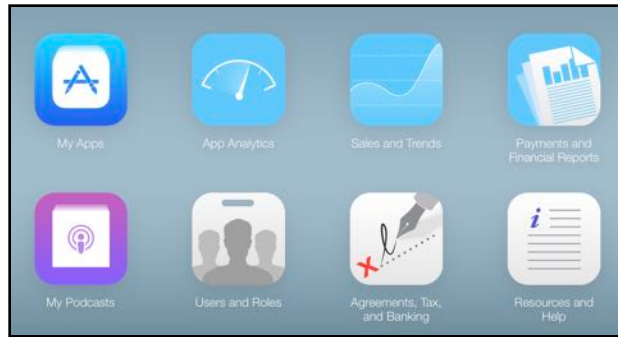
You don't have to download this provisioning profile, as Xcode will automatically fetch it from the Member Center when the time comes to sign the app.

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 administrivia, click on **My Apps** and then the **+** button and choose **New App**.

► Now is the time to enter some basic details about the app:

A screenshot of the 'New App' form in iTunes Connect. The form is titled 'New App' and has a 'Platforms' section with 'iOS' selected and 'tvOS' unselected. Below this is a 'Name' field with 'StoreSearch' entered. The 'Primary Language' is set to 'English'. The 'Bundle ID' is 'StoreSearch - com.razerware.StoreSearch'. The 'SKU' is 'APP4'. At the bottom are 'Cancel' and 'Create' buttons.

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.

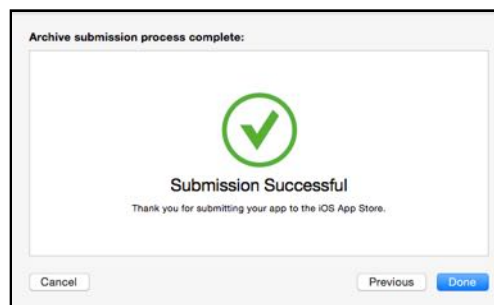
After fixing any of these issues, do **Product** → **Archive** again and validate the new archive.

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.

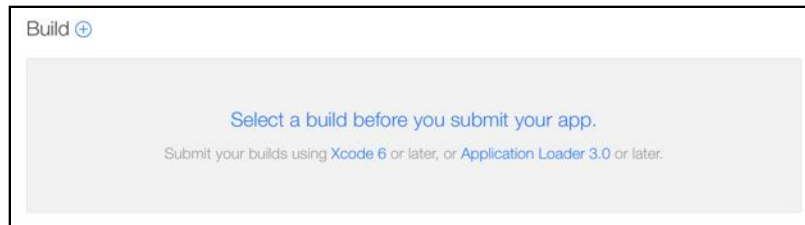
► In the Xcode Organizer, select the archive again and click **Upload to App Store**.

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!)

The end

Awesome, you've done it! You made it all the way through *The iOS Apprentice*. It's been a long journey but I hope you have learned a lot about iPhone and iPad programming, and software development in general. I had a lot of fun writing these tutorials and I hope you had a lot of fun reading them!

Because these tutorials are packed with tips and information you may want to go through them again in a few weeks, just to make sure you've picked up on everything!

The world of mobile apps now lies at your fingertips. There is a lot more to be learned about iOS and I encourage you to read the official documentation – it's pretty easy to follow once you understand the basics. And play around with the myriad of APIs that the iOS SDK has to offer.

Most importantly, go write some apps of your own!

Credits for this tutorial: The shopping cart from the app icon is based on a design from the Noun Project (thenounproject.com).

Want to learn more?

There are many great videos and books out there to learn more about iOS development. Here are some suggestions for you to start with:

- The iOS Developer Library has the full API reference, programming guides, and sample code: developer.apple.com/develop/
- The iOS Technology Overview gives a good introduction to what is possible on the iPhone and iPad: developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html
- Mobile Human Interface Guidelines (the “HIG”): developer.apple.com/ios/human-interface-guidelines/
- iOS App Programming Guide: developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Introduction/Introduction.html
- View Controller Programming Guide: https://developer.apple.com/library/content/featuredarticles/ViewControllerPGforiPhoneOS/#//apple_ref/doc/uid/TP40007457-CH2-SW1
- The WWDC videos. WWDC is Apple’s yearly developer conference and the videos of the presentations can be watched online at developer.apple.com/videos/. It’s really worth it!
- Myself and the rest of the raywenderlich.com team also have several other books for sale, including more advanced tutorials on iOS development and books about game programming on iOS. If you’d like to check these out, visit our store here: www.raywenderlich.com/store

Stuck?

If you get stuck, ask for help. Sites such as Stack Overflow (stackoverflow.com), the Apple Developer Forums (forums.developer.apple.com), and iPhoneDevSDK (www.iphonedevsdk.com/forum/) are great, and let’s not forget our own forums (forums.raywenderlich.com).

I often go on Stack Overflow to figure out how to write some code. I usually more-or-less know what I need to do – for example, resize a UIImage – and I could spend a few hours figuring out how to do it on my own, but chances are someone else already wrote a blog post about it. Stack Overflow has tons of great tips on almost anything you can do with iOS development.

However, don’t post questions like this:

“i am having very small problem i just want to hide load more data option in

tableView after finished loading problem is i am having 23 object in json and i am parsing 5 obj on each time at the end i just want to display three object without load more option.”

This is an actual question that I copy-pasted from a forum. That guy isn’t going to get any help because a) his question is unreadable; b) he isn’t really making it easy for others to help him.

Here are some pointers on how to ask effective questions:

- Getting Answers http://www.mikeash.com/getting_answers.html
- What Have You Tried? <http://mattgummell.com/what-have-you-tried/>
- How to Ask Questions the Smart Way <http://www.catb.org/~esr/faqs/smart-questions.html>

And that’s a wrap!

I hope you learned a lot through the *iOS Apprentice*, and that you take what you’ve learned to go forth and make some great apps of your own.

Above all, *have fun programming*, and let me know about your creations!

— Matthijs Hollemans