

Section 4: Store Search

The final section of the book covers iPad support in more detail via the *StoreSearch* app. *StoreSearch* shows you how to have separate custom screens both for specific orientations (landscape vs. portrait) as well as for specific platforms (iPhone vs. iPad). This section covers networking, working with remote API endpoints to fetch data needed by your app, and how to parse the fetched data.

If that wasn't enough, if this section also takes you through the full application life cycle - from developing the code, testing it, and submitting to Apple. So don't skip this section, thinking that you know all about iOS development after the last few sections!

Chapter 32: Search Bar

Chapter 33: Custom Table Cells

Chapter 34: Networking

Chapter 35: Asynchronous Networking

Chapter 36: URLSession

Chapter 37: The Detail Pop-up

Chapter 38: Polish the Pop-up

Chapter 39: Landscape

Chapter 40: Refactoring

Chapter 41: Internationalization

Chapter 42: The iPad

Chapter 43: Distributing the App

Table of Contents: Overview

Chapter 32: Search Bar	6
Chapter 33: Custom Table Cells	33
Chapter 34: Networking	62
Chapter 35: Asynchronous Networking.....	94
Chapter 36: URLSession	112
Chapter 37: The Detail Pop-up.....	142
Chapter 38: Polish the Pop-up	164
Chapter 39: Landscape	185
Chapter 40: Refactoring.....	221
Chapter 41: Internationalization	247
Chapter 42: The iPad	271
Chapter 43: Distributing the App.....	305

Table of Contents: Extended

Chapter 32: Search Bar	6
Create the project	8
Create the UI	12
Do fake searches	18
Create the data model	25
No results found	27
Chapter 33: Custom Table Cells	33
Custom table cells and nibs	34
Change the look of the app	47
Tag commits	51
The debugger	52
Chapter 34: Networking	62
Query the iTunes web service	63
Send an HTTP request	67
Parse JSON	72
Work with the JSON results	80
Sort the search results	90
Chapter 35: Asynchronous Networking	94
Extreme synchronous networking	95
The activity indicator	98
Make it asynchronous	105
Chapter 36: URLSession	112
Branch it	113
Put URLSession into action	115
Cancel operations	124
Search different categories	127
Download the artwork	132
Merge the branch	139

Chapter 37: The Detail Pop-up	142
The new view controller.....	143
Add the rest of the controls.....	150
Show data in the pop-up	157
Chapter 38: Polish the Pop-up.....	164
Dynamic Type	164
Gradients in the background.....	174
Animation!	179
Chapter 39: Landscape	185
The landscape view controller.....	186
Fix issues	196
Add a scroll view.....	201
Add result buttons.....	206
Paging	214
Download the artwork.....	217
Chapter 40: Refactoring	221
Refactor the search	222
Improve the categories	229
Enums with associated values	232
Spin me right round.....	238
Nothing found.....	242
The Detail pop-up	243
Chapter 41: Internationalization	247
Add a new language.....	248
Localize on-screen text	257
InfoPlist.strings	266
Regional settings	267
Chapter 42: The iPad	271
Universal apps	272
The split view controller	273

Improve the detail pane.....	280
Size classes in the storyboard	286
Your own popover	293
Send e-mail from the app.....	297
Landscape on iPhone Plus.....	301
Chapter 43: Distributing the App	305
Join the Apple Developer program	305
Beta testing	306
Submit to the App Store.....	323
The end	326

Chapter 32: Search Bar

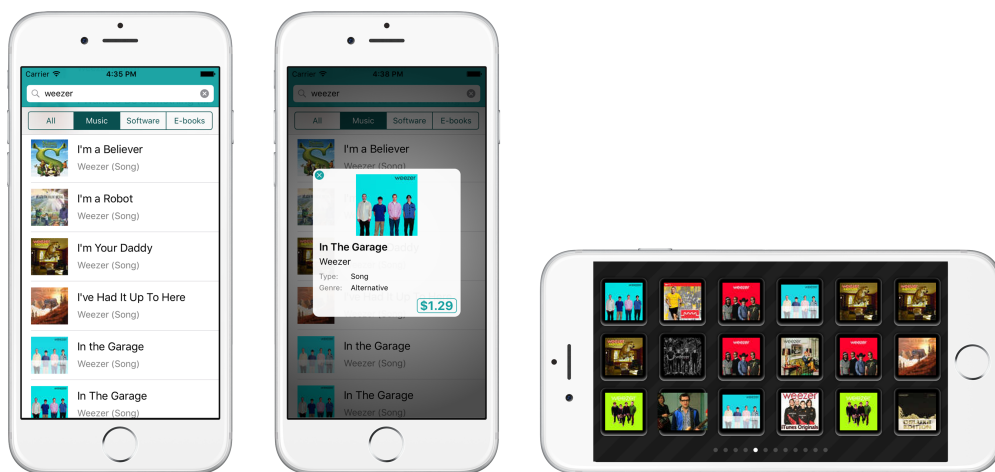
One of the most common tasks for mobile apps is to talk to a server on the Internet. It's beyond question: if you're writing mobile apps, you need to know how to upload and download data.

With this new app named *StoreSearch*, 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.

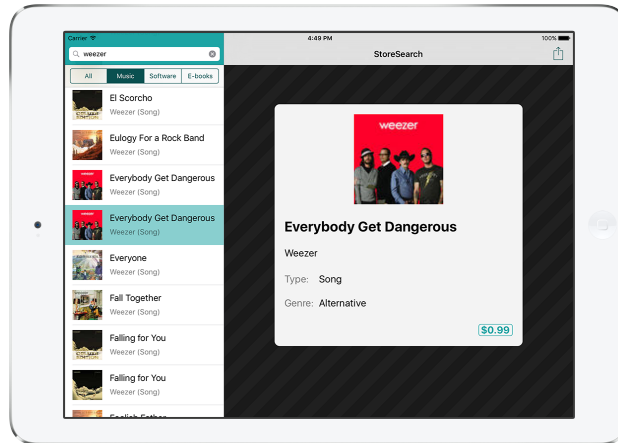
The finished app will look like this:



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 will also be an iPad version of the app with a custom UI for the iPad:



The app on the iPad

StoreSearch fills in the missing pieces and rounds off the knowledge you have obtained from developing the previous apps. 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.

In this chapter, you will do the following:

- **Create the project:** Create a new project for the new app. Set up version control using Git.
- **Create the UI:** Create the user interface for *StoreSearch*.
- **Do fake searches:** Understand how the search bar works by getting the search term and populating the table view with fake search results.
- **Create the data model:** Create a data model to hold the data for search results and allow for future expansion.
- **No data found:** Handle "no data" situations when doing a search.

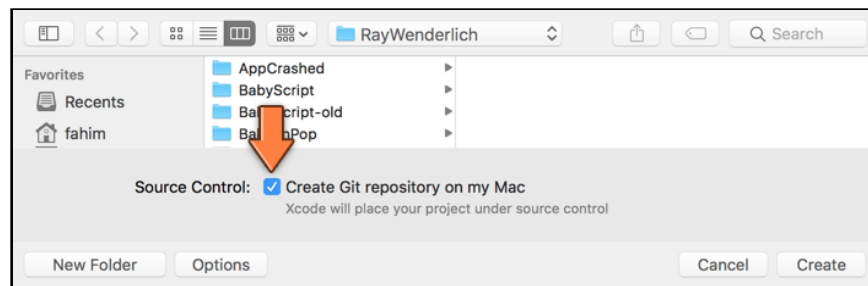
There's a lot of work ahead, so let's get started!

Create the project

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

- Product Name: **StoreSearch**
- Team: Default value
- Organization Name: your name
- Organization Identifier: com.yourname
- Language: **Swift**
- 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 **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 at the bottom-left of the dialog.

Git and version control

Git is a **version 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 the changes made to the project. Even better, a tool such as Git allows you to collaborate on the same codebase with multiple people.

Imagine what would happen if two programmers changed the same source file at the same time. Things would go horribly wrong! It's possible that your changes could 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 version control system such as Git, each programmer can work independently on the same files, without fear of undoing the work of others. Git is smart enough to automatically merge in all of the changes, and if there are any conflicting edits, it will let you resolve them manually.

Git is not the only version 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. Currently, Xcode has built-in support for both Git and Subversion (but Apple has indicated that the Subversion support will be deprecated in a future release.)

For *StoreSearch*, you will use some basic Git functionality. 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.

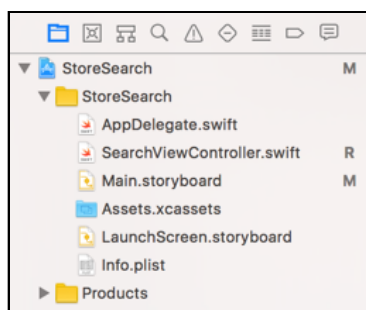
The first screen

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

► In the project navigator, select **ViewController.swift**, move your cursor over the ViewController class name and right-click to show the context menu. Select **Refactor** → **Rename...** from the menu and rename the class (and associated files and storyboard references) to `SearchViewController`.

► Run the app to make sure everything works. You should see a white screen with the status bar at the top.

Notice that the project navigator now shows **M** and **R** 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** → **Fetch and 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 renamed.

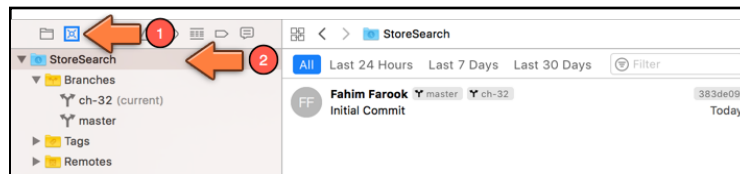
So what is a **commit**?

When you use a version 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.

Git version control

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

► Select the **Source Control** navigator from the Navigator pane and then click on the **project root** (the blue folder icon at the top) to see the project history:

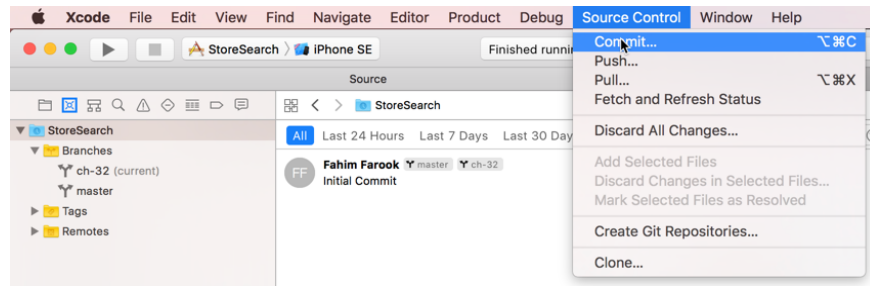


The history of commits for this project

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.

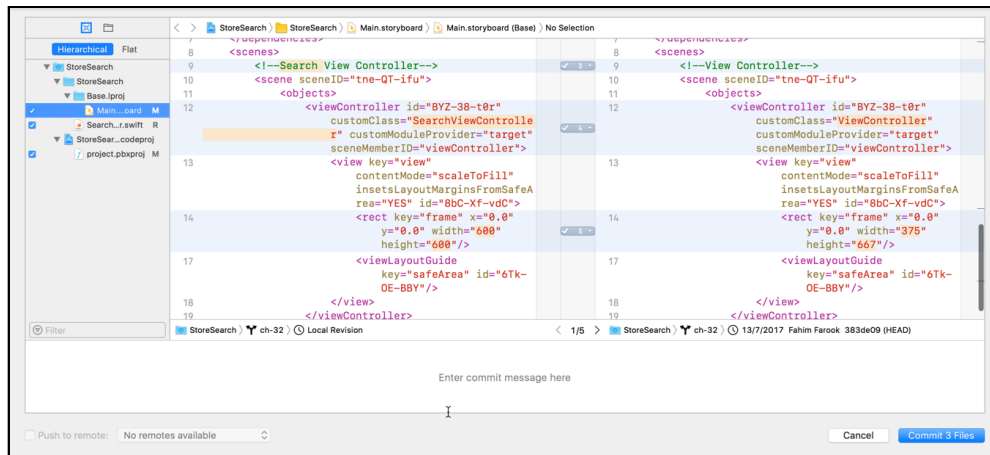
Note: Your Git history might not look the same as mine since mine also shows a *branch* called *ch-32*. Branches are a Git mechanism for working on the same codebase along different paths. You will learn more about Git branches in a later chapter. For the moment, just ignore the *ch-32* branch you see in the screenshot and know that if you don't have any other branches, that is fine - you are not supposed to :]

► Let's commit the change you just made. 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 code changes, just to make sure you're not committing anything you didn't intend to:



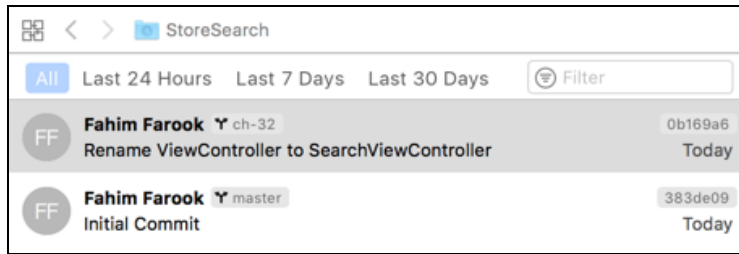
Xcode shows the changes you've made since the last commit

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** as the commit message.

► Press the **Commit 3 Files** button. You'll see that in the Project navigator the M and R icons are gone (at least until you make the next change).

The Source Control navigator should now show two commits (if it doesn't, click on a different branch in the list and then click on the root folder again).

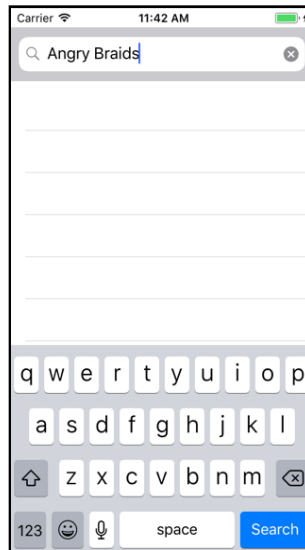


Your commit is listed in the project history

If you double-click a particular commit, 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 book you'll be a pro at it.

Create the UI

StoreSearch 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` (check the class definition in `SearchViewController.swift`, if you are not sure).

You are not required to use a `UITableViewController` as the base class for your view controller just because you have a table view in your UI. 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 - 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 - if you want that 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).

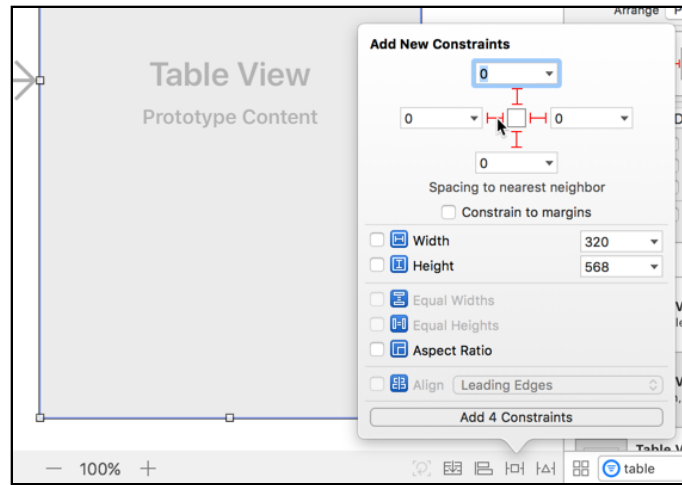
If your screen consists of just a UITableView, then it makes sense to make it a UITableViewController. But if you want to have other views as well, the more basic UIViewController is the option to go with.

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, or have the searchbar appear as part of the navigation bar, but for this app you will have it sitting above the table view.

Set up the storyboard

- 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** (*not* a Table View Controller) into the view controller.

► Make the Table View as big as the main view (320 by 568 points) and then use the **Add New Constraints** 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 for the *Bull's Eye* and *Checklists* apps. With Auto Layout you create **constraints** that determine how big the views are and where they go on the screen.

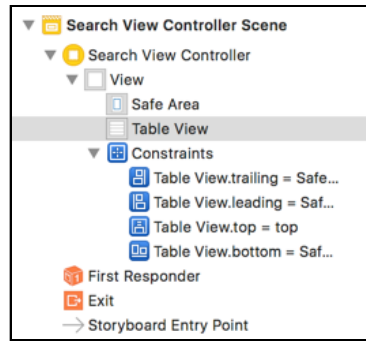
► First, uncheck **Constrain to margins** (if it is checked). Each screen has 16-point margins on the left and right (although you can change their size). When “Constrain to margins” is enabled you’re pinning to these margins. That’s no good here; you want to pin the Table View to the edge of the screen instead.

► In the **Spacing to nearest neighbor** section, select the red I-beams 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 the size of the device screen.

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

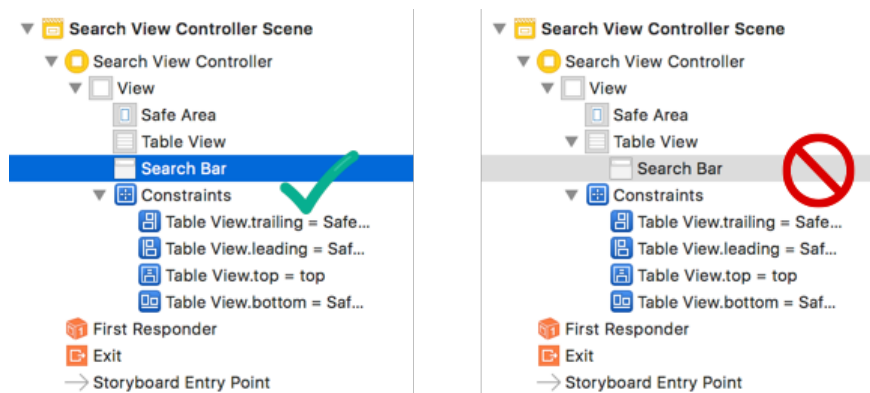
If you were successful, there should now be four blue bars surrounding the table view, one for each constraint. In the Document Outline there should also be a new Constraints section.



The new constraints in the Document Outline

► From the Object Library, drag a **Search Bar** on to 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.

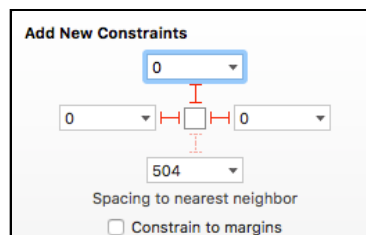
Make sure the Search Bar is not placed inside the table view. It should sit on the same level as the table view in the Document Outline:



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 Document Outline and drag it below the Table View.

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

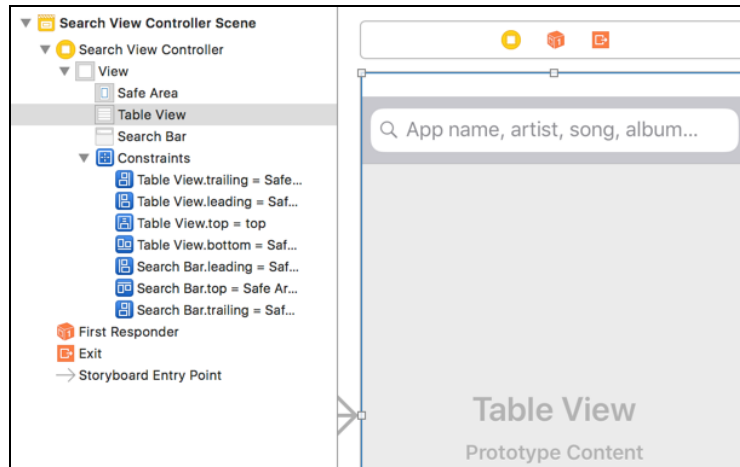


The constraints for the Search Bar

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

Connect to outlets

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!
```

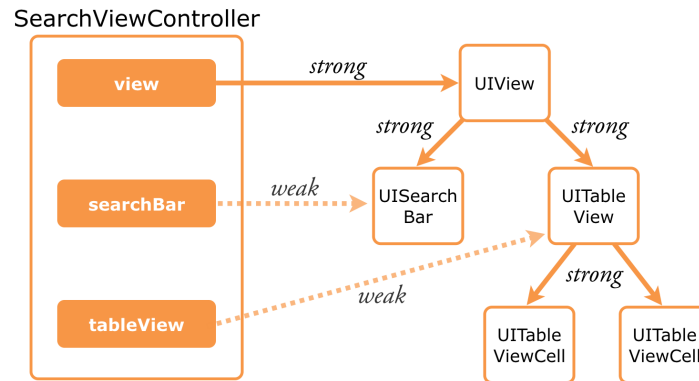
Recall that as soon as an object no longer has any strong references, it goes away – it is deallocated – and any weak references to it become nil.

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.

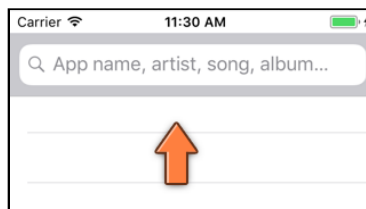


Outlets can be weak because the view hierarchy already has strong references

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

Table view content insets

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.

You could fix this in several different ways:

1. Change the table view's top layout constraint to match the search bar's bottom edge.
2. Make the Search Bar partially translucent to let the contents of the table cells shine through.
3. Use the table view's **content inset** attribute to allow for the area covered by the search bar.

You will go with option #3. Unfortunately, the content inset attribute is unavailable via Interface Builder. So, this has to be done from code.

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

```
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 - 20 points for the status bar and 44 points for the Search Bar.

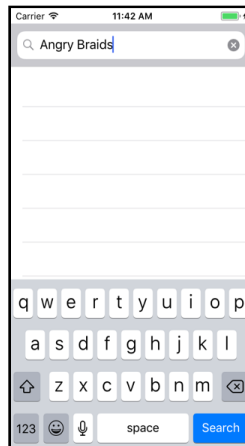
Now the first row will always be visible, and when you scroll the table view, the cells still go under the search bar. Nice.

Do fake searches

Before you implement the iTunes store searching, 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. Baby steps!

- Run the app. If you tap the search bar, the on-screen keyboard will appear, but it won't do anything when you type in a search term and 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 a search bar delegate

► Add the following to the bottom of **SearchViewController.swift**, after 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.

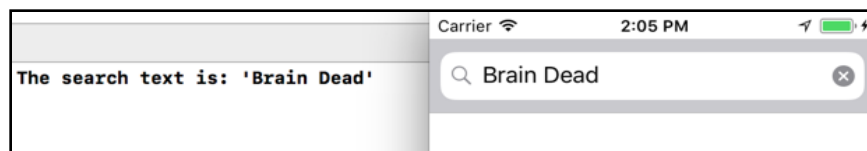
The `UISearchBarDelegate` protocol has a method `searchBarSearchButtonClicked(_:)` that is invoked when the user taps the Search button on the keyboard. You will implement this method to put some fake data into the table. (Later, you'll make this method send a network request to the iTunes store to find 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!)

At the moment, all the new code does is to output the search term from the search bar to the Xcode Console.

Tip: I always put strings 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, **Control-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 Xcode Console

Show fake results

- Add the following new (and empty) extension to **SearchViewController.swift**:

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

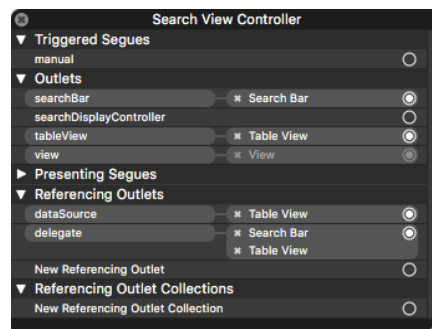
The above extension will handle all the table view related delegate methods. You could certainly have added them as two separate extensions if you liked, but I prefer to keep all the table view delegate related code in one place.

Adding the `UITableViewDataSource` and `UITableViewDelegate` protocols wasn't necessary for the previous apps because you used a `UITableViewController` in each case. `UITableViewController` already conforms to these protocols by necessity.

`SearchViewController` however, is a regular view controller and therefore you have to hook up the data source and delegate protocols yourself.

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

In case you're wondering how you connected something to a delegate property in Search View Controller 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 Control-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.

► Add the minimum code you need to comply:

```
extension SearchViewController: UITableViewDataSource,
    UITableViewDelegate {
    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 - for example, this works fine for `UISearchBarDelegate`.

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 apps? Look closely...

Answer: They don't have the `override` keyword.

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 `tableView(_: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]()
```

The search bar delegate method will put some fake data into this array and then display it using 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 replace the contents of `searchResults` property with it. This is done each time the user performs a search. If there was already a previous array of results, then that 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 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.

► Replace the methods in the table view delegate extension with the following:

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

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

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

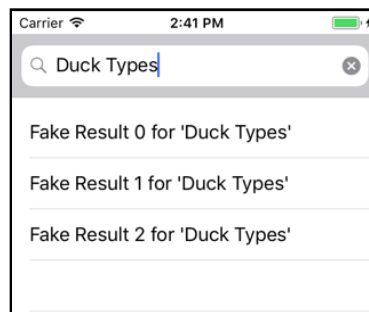
if cell == nil {
    cell = UITableViewCell(style: .default,
        reuseIdentifier: cellIdentifier)
}

cell.textLabel!.text = searchResults[indexPath.row]
return cell
}
```

All of the above code should be pretty familiar to you by now. You simply return the number of rows to display based on the contents of the `searchResults` array and you create a `UITableViewCell` by hand to display the table rows.

► Run the app. If you search for anything, a few 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

Dismiss keyboard on search

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

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

```
searchBar.resignFirstResponder()
```

This tells the `UISearchBar` that it should no longer listen for keyboard input. As a result, the keyboard will hide itself until you tap on 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**.

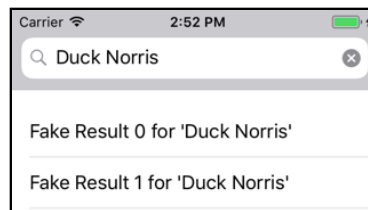
Extend search bar to status area

The search bar still has an ugly white gap above it for the status area. It would look a lot better if the status bar area was unified with the search bar. There's a delegate method for `UINavigationController` and `UISearchBar` items which allows the item to indicate its top position.

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

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

Now the app looks way better:



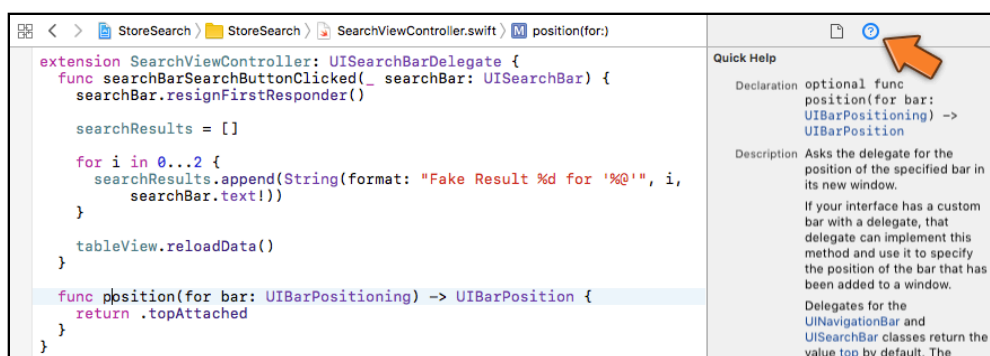
The search bar is “attached” to the top of the screen

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

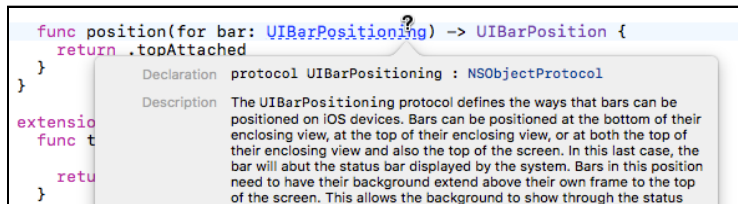
Xcode comes with a big library of documentation for developing iOS apps. Basically everything you need to know is in here. Learn to use the Xcode documentation browser – it will become your best friend!

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

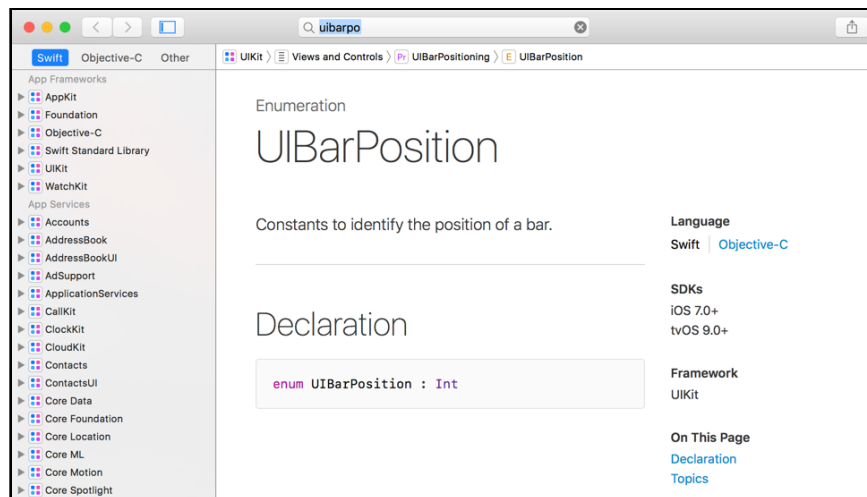


Simply have the **Quick Help inspector** open (the second tab in the inspector pane) and it will show context-sensitive help. Put the text cursor on the item you want to know more about and the inspector will provide a summary. You can click any of the blue text to jump to the full documentation.

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:



Create 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.

The SearchResult class

➤ 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. You'll add several others in a bit.

In SearchViewController you need to modify the searchResults array to hold instances of SearchResult.

➤ In **SearchViewController.swift**, change the declaration of the property:

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

➤ Next, 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)  
}
```

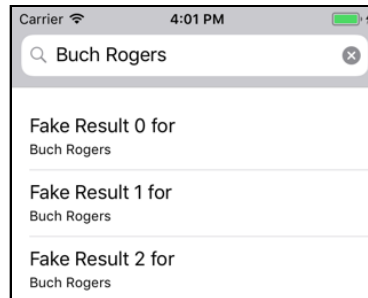
This creates an instance of the SearchResult object and simply puts some fake text into its name and artistName properties. Again, you do this in a loop because just having one search result by itself is a bit lonely.

➤ At this point, tableView(_:cellForRowAt:) still expects the array to contain strings. So, update that method:

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

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

► Run the app; it should look like this:



Fake results in a subtitle cell

No results found

When you add search functionality to your apps, you 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, there is no reason why you cannot fake the last scenario as well.

Handle not getting any results

In defense of good taste, the app will return 0 results when a 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 prevents the creation of any `SearchResult` objects if the text is equal to "justin bieber".

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

At this point, you don't know if the search failed, or if there were no results. 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 last part of `tableView(_:cellForRowAt:)` to:

```

if cell == nil {
    ...
}
// New code
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
}
// End of new code
return cell

```

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

► Change `tableView(_:numberOfRowsInSection:)` to:

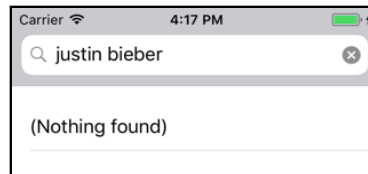
```

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

```

Now, if there are no results, the method 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...

Handle no results when app starts

Unfortunately, the text “Nothing found” also appears initially when the user has not searched 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:

- Change `searchResults` to 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 whether a search has been done yet or not.

It may be tempting to choose the optional, but it’s best to avoid optionals if you can. They complicate the logic, 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 on your own, and compare the differences. It’ll be a great exercise!)

► Still in **SearchViewController.swift**, add a 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 // Add this line  
    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 you'll see a much better way to handle this using an enum – and it will blow your mind!)

Selection handling

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

► To fix that, add the following method to the table view delegate 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 when you have 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 a bit.)

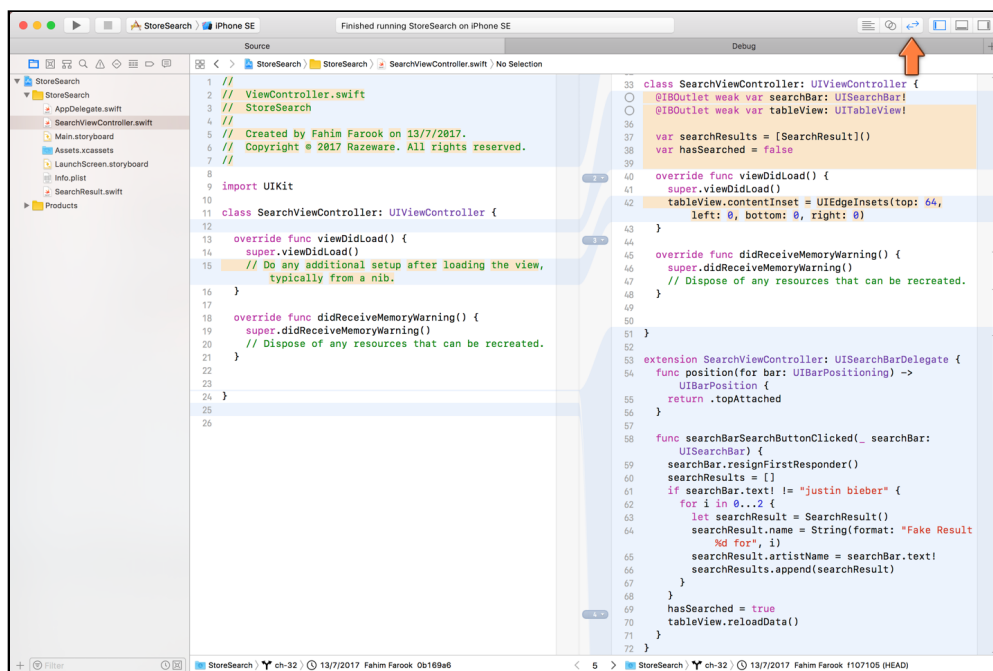
► This is a good time to commit your changes. Go to **Source Control** → **Commit** (or press the **⌘+Option+C** keyboard shortcut).

Make sure all the files are selected/checked in the list on the left, 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”.

Versions editor

If you ever want to look back through your commit history, you can do that from the Source Control navigator (as you learnt how at the beginning of this chapter) or from the **Version editor**, pictured below:



Viewing revisions in the Version editor

You switch to the Version editor using the relevant toolbar button on the top right of the Xcode window.

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

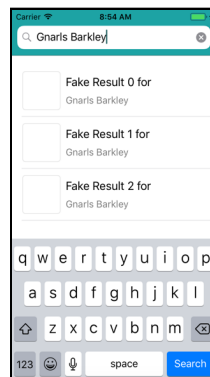
The app isn't very impressive yet, but you've laid the foundation for what is to come. You have a search bar and know how to take action when the user presses the Search button. The app also has a simple data model that consists of an array with `SearchResult` objects, and it can display these search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

You can find the project files for this chapter under **32 – Search Bar** in the Source Code folder.

Chapter 33: Custom Table Cells

Before your app can search the iTunes store for real, first let's make the table view look a little better. Appearance does matter when it comes to apps!

Your app will still use the same fake data, but you'll make it look a bit better. This is what you're going to make in this chapter:



The app with better looks

In the process, you will learn the following:

- **Custom table cells and nibs:** How to create, configure and use a custom table cell via nib file.
- **Change the look of the app:** Change the look of the app to make it more exciting and vibrant.
- **Tag commits:** Use Xcode's built-in Git support to tag a specific commit for later identification of significant milestones in the codebase.
- **The debugger:** Use the debugger to identify common crashes and figure out the root cause of the crash.

Custom table cells and nibs

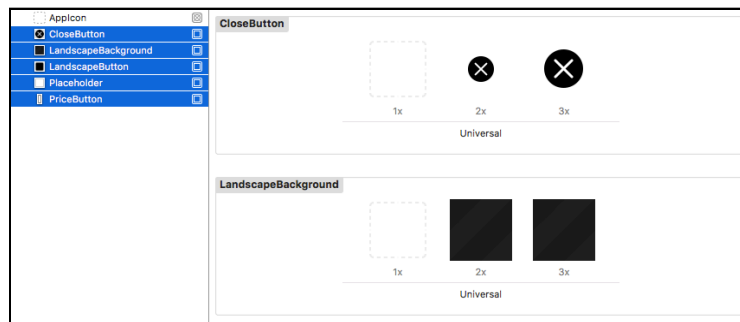
For the previous apps, you used prototype cells to create your own table view cell layouts. That works great, but there's another way. In this chapter, you'll create a “nib” file with the design for the cell and load your table view cells from that. The principle is very similar to prototype cells.

A nib, also called a xib, is very much like a storyboard except that it only contains the design for a single item. That item 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.

Add assets

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

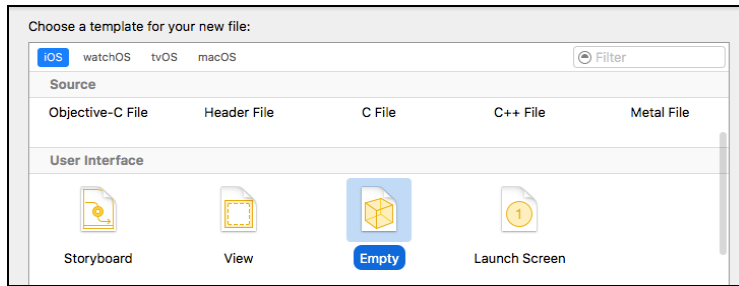


Imported images in the asset catalog

Each of the images comes in two versions: 2x and 3x. There are no low-resolution 1x devices that can run the latest version of iOS. So there's no point in including 1x images.

Add a nib file

► 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 empty nib.



Adding an empty nib to the project

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

Open **SearchResultCell.xib** and you will see an empty canvas.

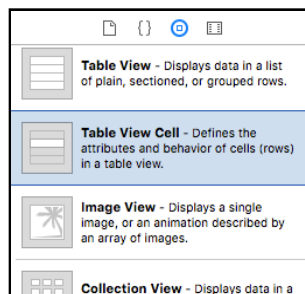
Xib or nib

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*, from the old NeXT platform 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 jargon.)

➤ 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 larger devices/screens.

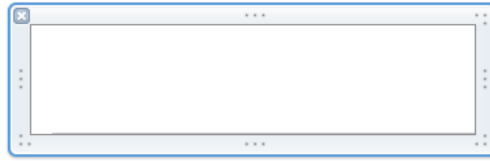
➤ From the Object Library, drag a new **Table View Cell** on to the canvas:



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.

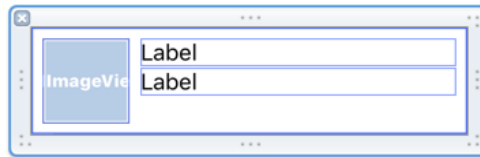
The cell now looks like this:



An empty table view cell

Note: Sometimes, you might have a blue bounding rectangle for the cell which is slightly offset from the actual cell's location. This is an Interface Builder bug. If this happens to you, simply switch to some other file and then switch back to the SearchResultCell.xib - all should be well at this point.

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



The design of the cell

Note: If you get blue rectangles around each item like above (or would like to get the rectangles to see the full bounds of each item), then use the **Editor** → **Canvas** → **Show Bounds Rectangles** menu item to toggle the bounds rectangles on/off.

- Position the Image View at X:16, Y:10, Width:60, Height:60.
- Set the **Text** of the first label to **Name**, **Font** to **System 18**, X:84, Y:16, **Width**:220, **Height**:22.
- Set the **Text** for the second label to **Artist Name**, **Font** to **System 15*, **Color** to **black with 50% opacity**, X:84, Y:44, **Width**:220, **Height**:18..

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

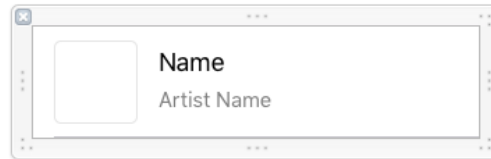
► The Table View Cell itself needs to have a reuse identifier. You can set this in the **Attributes inspector** to **SearchResultCell**.

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 should now look like this:



The cell design with placeholder image

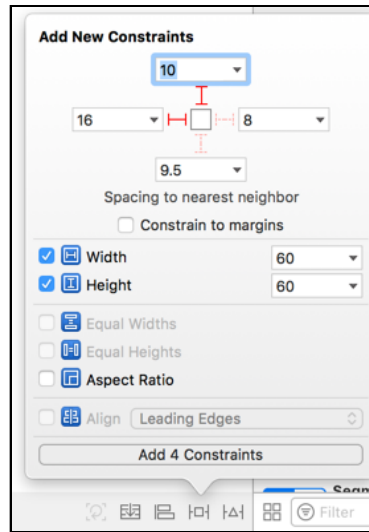
You're not done yet. The design for the cell is only 320 points wide but there are iOS devices with screens wider than that. The cell itself will resize to accommodate those larger screens, but the labels won't, potentially causing their text to be cut off. You'll have to add some Auto Layout constraints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

Set up Auto Layout constraints

When setting up Auto Layout constraints, it's best to start from one edge (like the top left for left-to-right screens - remember there are also screens which can be right-to-left) and to work your way left and down. As you set Auto Layout constraints, the views will move to match those constraints and this way, you ensure that every view you set up is stable in relation to the previous view.

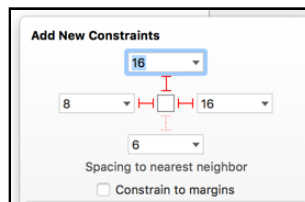
If you randomly set up layout constraints for views, you'll see your views moving all over the place and you might not remember after a while where you originally had any view placed.

► Select the **Image View** and open the **Add New Constraints** menu. Uncheck **Constrain to margins** and pin the Image View to the **top** and **left** sides of the cell. Also give it **Width** and **Height** constraints so that its size is always fixed at 60 by 60 points:



The constraints for the Image View

- Click **Add 4 Constraints** to actually add the constraints.
- Select the **Name** label and again use the **Add New Constraints** menu. Uncheck **Constrain to margins** and select the **top**, **left**, and **right** pins (but not the bottom one):



The constraints for the Name label

- Click **Add 3 Constraints**.
- Finally, pin the **Artist Name** label to the **left**, **right**, and **bottom** (again without constraining to margins).

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

Register nib file for use in code

- In **SearchViewController.swift**, add these lines to the end 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 with just one statement:

```
let cellIdentifier = "SearchResultCell"
var cell: UITableViewCell! = tableView.dequeueReusableCell(
    withIdentifier: cellIdentifier)
if cell == nil {
    cell = UITableViewCell(style: .subtitle,
                          reuseIdentifier: cellIdentifier)
}
```

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 custom cell design, you cannot 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 built-in labels get in the way of your own labels.

In this case, 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 custom `UITableViewCell` subclass

➤ Add a new file to the project using the **Cocoa Touch Class** template. Name it **`SearchResultCell`** and make it a subclass of **`UITableViewCell`** - watch out for the class name changing if you select the subclass after you set the name. ("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.)

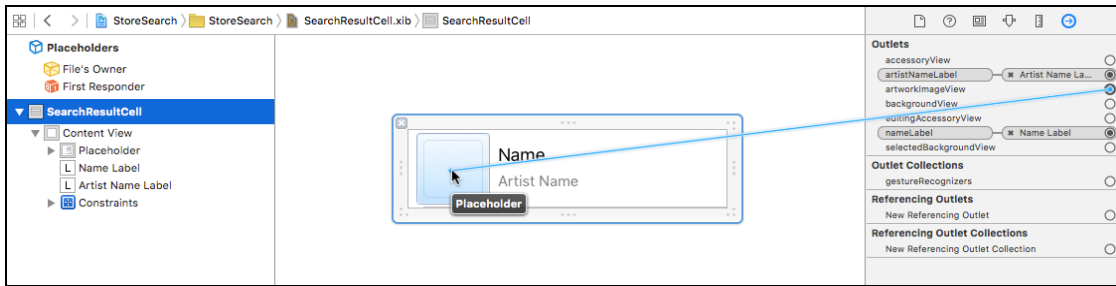
➤ In the **Identity inspector**, change its class from "`UITableViewCell`" to **`SearchResultCell`**.

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 Control-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.

Use custom table view cell in app

► In **SearchViewController.swift**, change `cellForRowAt` to:

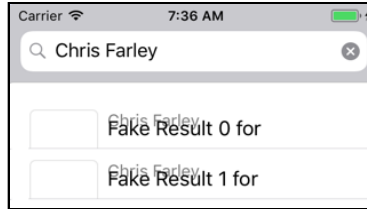
```
func tableView(_ tableView: UITableView,
               cellForRowAt indexPath: IndexPath) -> UITableViewCell {

    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.

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



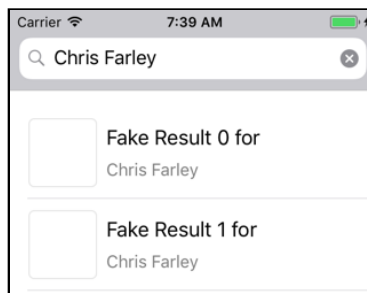
Uh oh...

The problem is that these table 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, and you can do it either in storyboards (via the Size inspector for the table view) or in code. Let's try code...

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

```
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.

Use a constant for table cell identifier

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 the identifier "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 within the class definition:

```
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.

► In **`SearchViewController.swift`**, replace the string `"SearchResultCell"` with `TableViewCellIdentifiers.searchResultCell`.

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

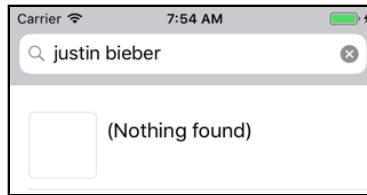
```
override func viewDidLoad() {  
    . . .  
    let cellNib = UINib(nibName:  
        TableViewCellIdentifiers.searchResultCell, bundle: nil)  
    tableView.register(cellNib, forCellReuseIdentifier:  
        TableViewCellIdentifiers.searchResultCell)  
    . . .  
}
```

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

► Run the app to make sure everything still works.

A new “No results” cell

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

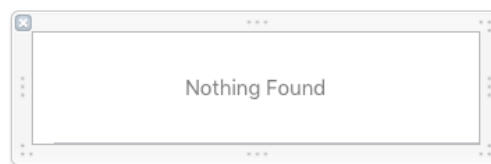


The Nothing Found label now looks like this

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

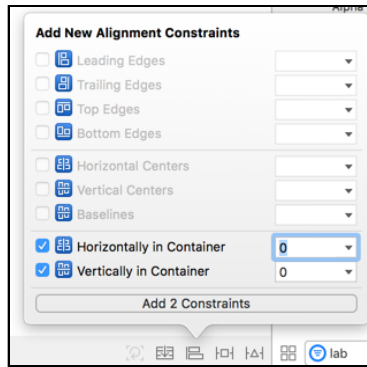
- Add another nib file to the project. Again this will be an **Empty** nib. Name it **NothingFoundCell.xib**.
- Drag a new **Table View Cell** on to the canvas. Set its **Width** to 320, its **Height** to 80 and give it the reuse identifier **NothingFoundCell**.
- Drag a **Label** into the cell and give 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

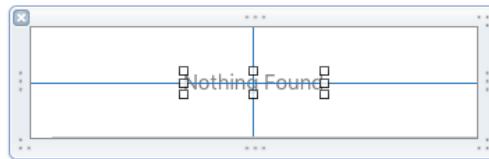
In order to keep the text centered on all devices, select the label and open the **Align menu**:



Creating the alignment constraints

- Choose **Horizontally in Container** and **Vertically in Container** and click **Add 2 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 a new reuse identifier 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 registering the nib:

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

This also requires you to change `let cellNib` two lines up to `var` because you're re-using the `cellNib` local variable.

- And finally, change `tableView(_:cellForRowAt:)` to:

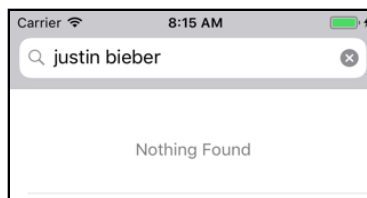
```
func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {

    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 since there is nothing to configure for that cell.

- 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 larger screen devices. 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.”

Change 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**:

```
// MARK:- Helper Methods
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 fell swoop.

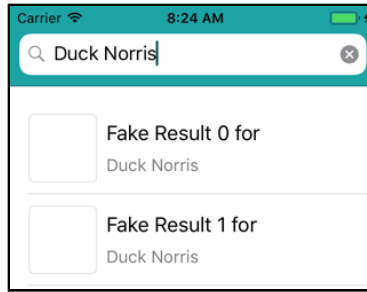
The `UIColor(red:green:blue:alpha:)` method makes a new `UIColor` object based on the RGB and alpha color components that you specify.

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() // Add this line
    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 “Passing the context” section for the *MyLocations* app.

Change the row selection color

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

As you learnt with *MyLocations*, 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.

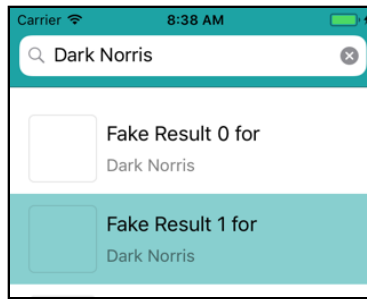
For example, in `init?(coder)` the `nameLabel` and `artistNameLabel` outlets will still be `nil` but in `awakeFromNib()` they will be properly hooked up to their `UILabel` objects. So, if you wanted to do something with those outlets in code, you'd need to do that in `awakeFromNib()`, not in `init?(coder)`.

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()` - it 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, do a search and tap a row, it should look like this:



The selection color is now green

Add app icons

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

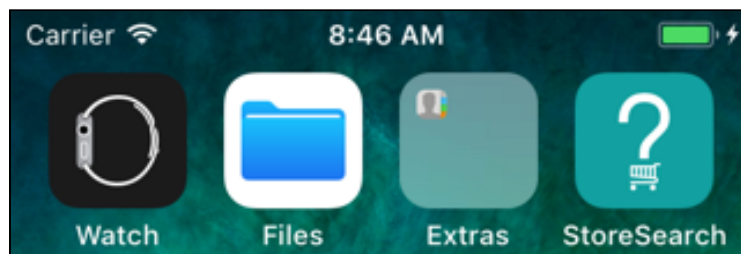
- Open the asset catalog (**Assets.xcassets**) and select the **AppIcon** group.
- Drag the images from the **Icon** folder from the Resources folder into the matching slots.

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 now has a nice new icon:



The app icon

Show keyboard on app launch

One final user interface tweak I'd like to make is that the keyboard should 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**:

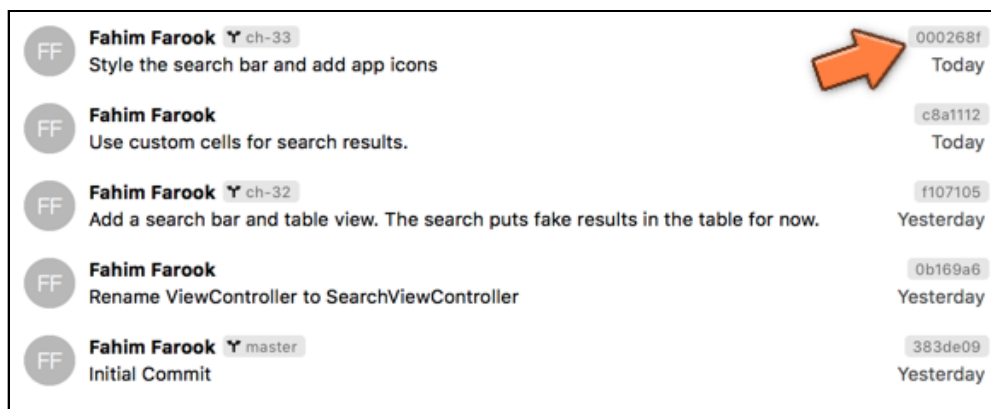
```
searchBar.becomeFirstResponder()
```

As you are aware from the *Checklists* app, `becomeFirstResponder()` will give `searchBar` the "focus" and show the keyboard. Anything you type will end up in the search bar.

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

Tag commits

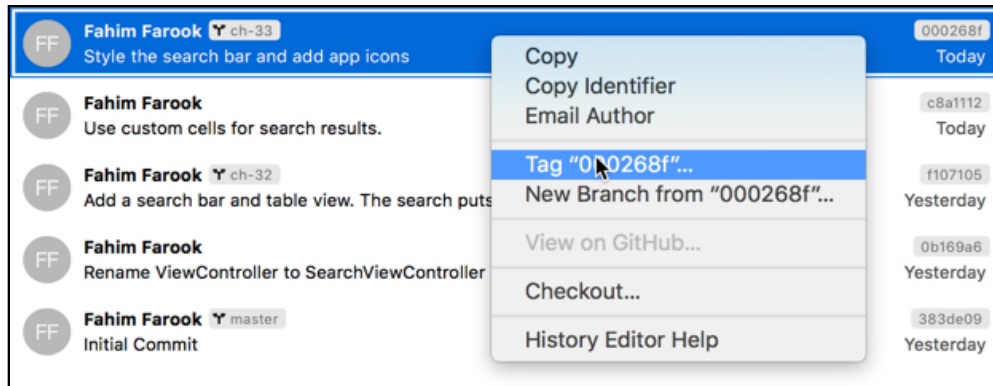
If you look through the various commits you've made so far, you'll notice a bunch of strange numbers, such as "000268f":



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

Those are internal numbers (known as the "hash") that Git uses to uniquely identify commits. Such numbers aren't very memorable (or useful) for us humans, so Git also allows you to "tag" a certain commit with a more friendly label.

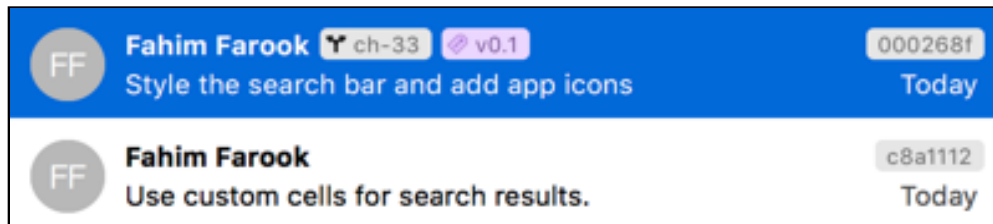
- Tagging a commit in Xcode is as simple as selecting the commit in the Source Control navigator view, right-clicking to get the context menu and selecting the **Tag** option.



Tagging a commit in Xcode

► Enter "v0.1" as the **Tag**, and an optional message describing what this particular tag encompasses. Then click **Create** to create the tag.

You can see the new tag in the Source Control navigator view:



The new tag in Xcode

Xcode works quite well with Git but you might want an app with more power to do complex Git operations. If you do, you'll probably need to learn how to use the Terminal or get a tool such as SourceTree (free on the Mac App Store).

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

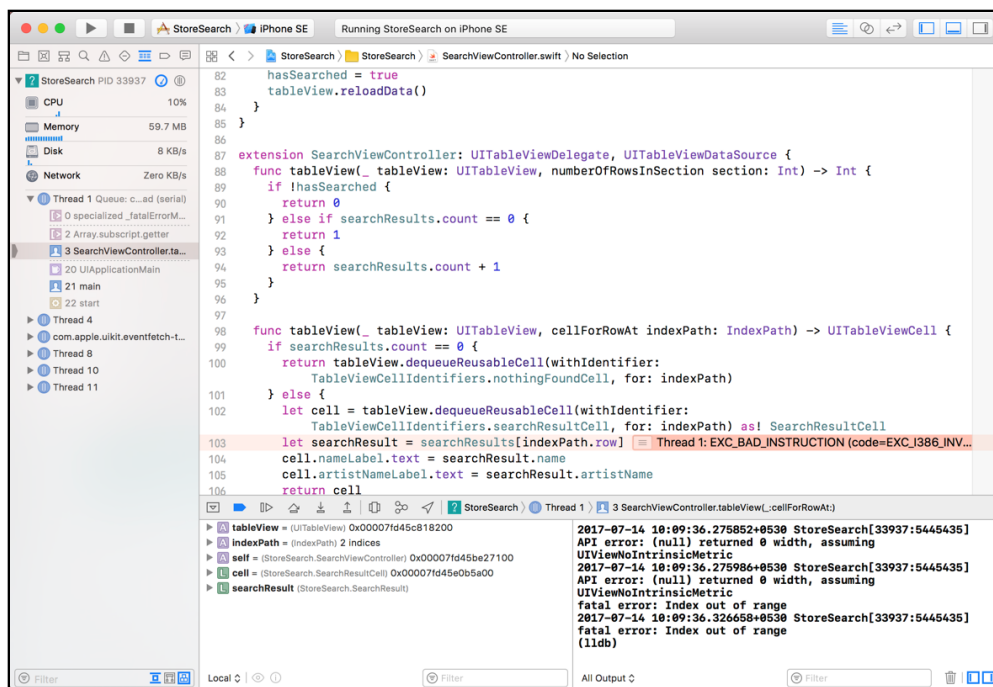
Index out of range bug

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

► 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 // This line changes
    }
}
```

► 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 Console:

```
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 out the offending line:

```
102 let cell = tableView.dequeueReusableCell(withIdentifier:
    TableViewCellIdentifiers.searchResultCell, for: indexPath) as! SearchResultCell
103 let searchResult = searchResults[indexPath.row] Thread 1: EXC_BAD_INSTRUCTION (code=EXC_I386_INV...
104 cell.nameLabel.text = searchResult.name
```

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 and there are several ways to do this.

The one we’ll look at here is to use the debugger’s command line interface, like a hacker whiz kid from the movies :]

► In the Xcode Console, after the **(lldb)** prompt, type **p indexPath.row** and press enter:

```
fatal error: Index out of range
(lldb) p indexPath.row
(Int) $R0 = 3
(lldb)
```

All Output ⌵ Filter

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 (if you use the auto complete functionality, do note that both `searchResult` - without the "s" at the end - and `searchResults` are choices. Make sure to select the correct one.):

```
(lldb) p indexPath.row
(Int) $R0 = 3
(lldb) p searchResults
([StoreSearch.SearchResult]) $R1 = 3 values {
  [0] = 0x00007f8a31407d50 (name = "Fake Result 0 for", artistName = "weezer")
  [1] = 0x00007f8a314827d0 (name = "Fake Result 1 for", artistName = "weezer")
  [2] = 0x00007f8a31482810 (name = "Fake Result 2 for", artistName = "weezer")
}
```

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.

Storyboard outlet bug

► Restore `numberOfRowsInSection` to its previous state and then add a new outlet property to **SearchViewController.swift**:

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

► Open the storyboard and **Control-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.)

► Delete the `searchBar2` outlet property from **SearchViewController.swift** (in the source code - not the storyboard).

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 0x7fb83ec09bf0>
setValue:forUndefinedKey:]: this class is not key value coding-compliant
for the key searchBar2.'
*** First throw call stack:
(
    0  CoreFoundation          0x0000000111da1c7b __exceptionPreprocess +
171
    . . .
```

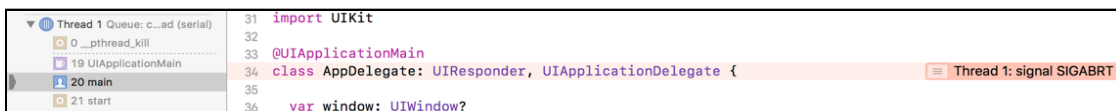
The first part of this message is very important: it tells you that the app was terminated because of an “NSUnknownKeyException”. On some platforms, exceptions are a commonly used error handling mechanism, but on iOS this is always a fatal error and the app is forced to halt.

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:

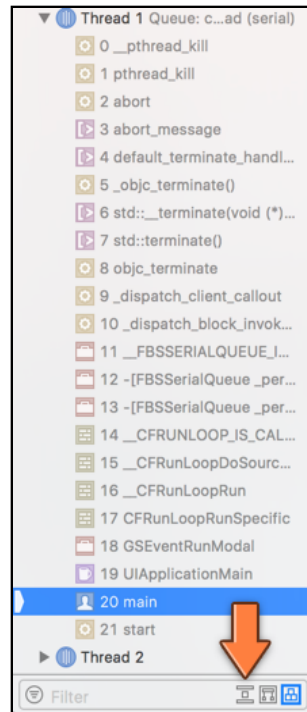


Crash in AppDelegate?

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.

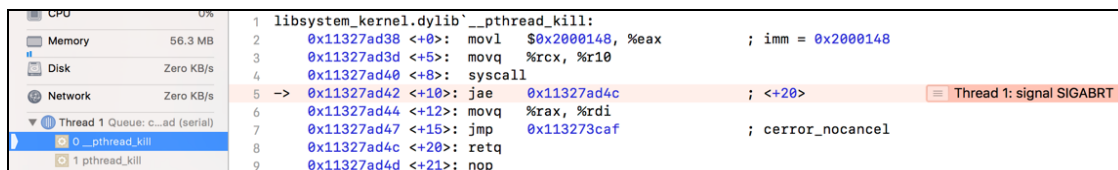
► Click the left-most icon at the bottom of the Debug navigator to see more info.



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

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

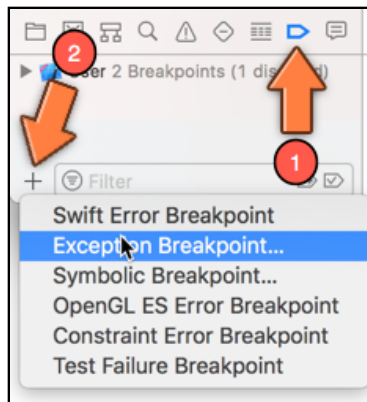
A **breakpoint** is a special marker in your code that will pause the app execution and launch the debugger.

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 book, but you can read more about it in the Debugging section of Apple's developer support site: developer.apple.com/support/debugging. Or, you can check the **Debug your app** topic under Xcode's **Help** → **Xcode Help** menu option.

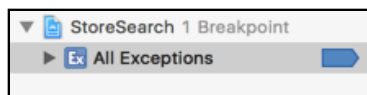
You are going to set a special breakpoint that is triggered whenever a fatal exception occurs. This will halt the program just as it is about to crash, which should give you more insight into what is going on.

► Switch to the **Breakpoint navigator** and click the + button at the bottom to add an Exception Breakpoint:



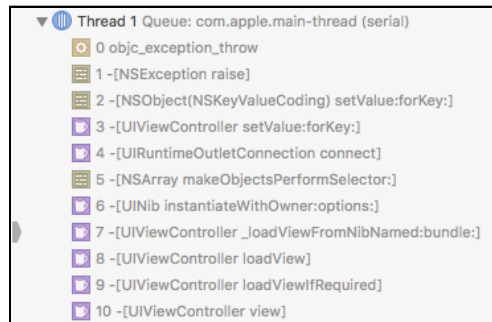
Adding an Exception Breakpoint

This will add a new breakpoint:



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 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 in 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 Console 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 by simply selecting the breakpoint and clicking on the dark blue arrow. If the arrow goes from dark blue to a pale blue, it is disabled.

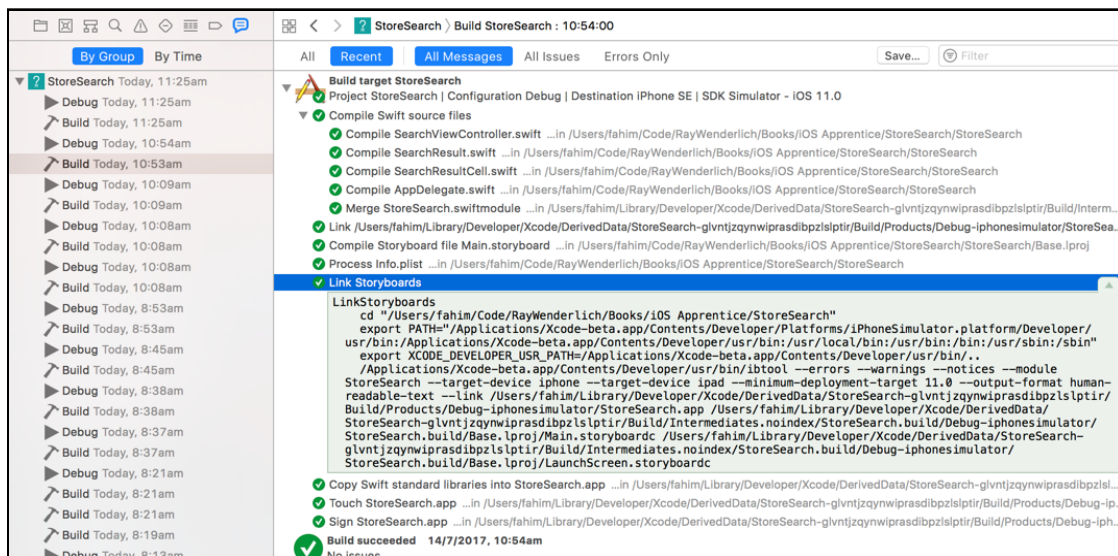
To summarize:

- 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!), add an Exception Breakpoint to get more info.
- If the app crashes with a `SIGABRT` but there is no error message, then disable any Exception Breakpoints you may have 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... eventually.)
- 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 points 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 **Report navigator**. It’s the last tab in the navigator pane.



The Report 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, select the item and click the little detail 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.

You can find the project files for this chapter under **33 – Custom Table Cells** in the Source Code folder.

Chapter 34: Networking

Now that the preliminaries are out of the way, you can finally get to the good stuff: adding networking to the app so that you can download actual data from the iTunes Store!

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 *StoreSearch*, but you will use that free web service to perform searches.

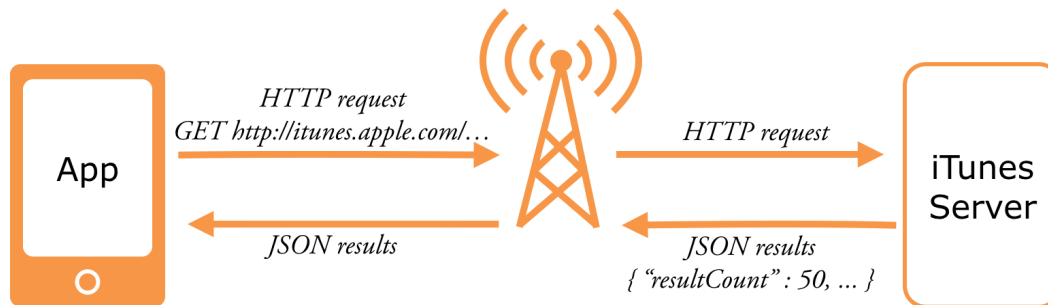
In this chapter you will learn the following:

- **Query the iTunes web service:** An introduction to web services and the specifics about querying Apple's iTunes Store web service.
- **Send an HTTP request:** How to create a proper URL for querying a web service and how to send a request to the server.
- **Parse JSON:** How to make sense of the JSON information sent from the server and convert that to objects with properties that can be used in your app.
- **Sort the search results:** Explore different ways to sort the search results alphabetically so as to write the most concise and compact code.

Query the iTunes web service

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 process. All you need to know is that you’re using HTTP.

HTTP is the 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 - if you don’t know what that means, don’t worry about it for now...) 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 will download a file. If you open the file in a text editor, it should contain something like this:

```
{
  "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", "artistViewUrl":"https://itunes.apple.com/us/artist/metallica/
id3996865?uo=4", "collectionViewUrl":"https://itunes.apple.com/us/album/
enter-sandman/id579372950?i=579373079&uo=4", "trackViewUrl":"https://
itunes.apple.com/us/album/enter-sandman/id579372950?i=579373079&uo=4",
"previewUrl":"http://a38.phobos.apple.com/us/r30/Music7/v4/bd/fd/e4/
bdfde4e4-5407-9bb0-e632-edbf079bed21/
mzaf_907706799096684396.plus.aac.p.m4a", "artworkUrl30":"http://
is1.mzstatic.com/image/thumb/Music/v4/0b/9c/
d2/0b9cd2e7-6e76-8912-0357-14780cc2616a/source/30x30bb.jpg",
"artworkUrl60":"http://is1.mzstatic.com/image/thumb/Music/v4/0b/9c/
d2/0b9cd2e7-6e76-8912-0357-14780cc2616a/source/60x60bb.jpg",
"artworkUrl100":"http://is1.mzstatic.com/image/thumb/Music/v4/0b/9c/
d2/0b9cd2e7-6e76-8912-0357-14780cc2616a/source/100x100bb.jpg",
"collectionPrice":9.99, "trackPrice":1.29,
"releaseDate":"1991-07-29T07:00:00Z",
"collectionExplicitness":"notExplicit",
"trackExplicitness":"notExplicit", "discCount":1, "discNumber":1,
"trackCount":12, "trackNumber":1, "trackTimeMillis":331560,
"country":"USA", "currency":"USD", "primaryGenreName":"Metal",
"isStreamable":true},
. . .
```

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 being fast replaced by 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 file with a **.json** extension 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). Go ahead and copy-paste these URLs in your browser and see what happens.

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 each item 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 `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 some 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** - it 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.

Send an HTTP request

In order to query the iTunes Store web service, the very first thing you must do is send an HTTP request to the iTunes server. This involves several steps such as creating a URL with the correct search parameters, sending the request to the server, getting a response back etc.

You'll take these step-by-step.

Create the URL for the request

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

```
// MARK:- Private Methods
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 a URL string by placing the search text behind the “term=” parameter, and then turns this string into a URL object.

Because `URL(string:)` is a failable initializer, 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, Apple recommends that 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 use HTTPS, 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 on you'll learn how to do this because the artwork images are hosted on a server that does not support 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 created 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 an `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 only 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 that is a single word, for example “metallica” (or one of your other favorite metal bands), and press the Search button.

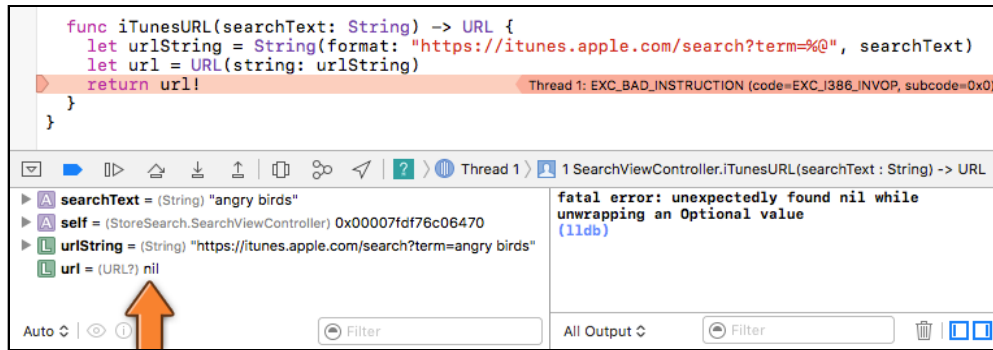
Xcode should now show this in its Debug pane:

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

That looks good.

► Now type in a search term with one or more spaces, like “angry birds”, into the search box.

Whoops, the app crashes!



The crash after searching for “angry birds”

Look at the left-hand pane, the **Variables view**, 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 encodedText = searchText.addingPercentEncoding(
        withAllowedCharacters: CharacterSet.urlQueryAllowed)!
    let urlString = String(format:
        "https://itunes.apple.com/search?term=%@", encodedText)
    let url = URL(string: urlString)
    return url!
}
```

This calls the `addingPercentEncoding(withAllowedCharacters:)` method to create a new string where all the special characters are escaped, and you use that string 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 with 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.

Perform the search request

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.localizedDescription)")  
        return nil  
    }  
}
```

The meat of this method is the call to `String(contentsOf:encoding:)` which 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 branch and the error variable will contain more details

about the error. If this happens, you print out a user-understandable form of the error and return nil to signal that the request failed.

► Add the following lines to `searchBarSearchButtonClicked(_:)`, after the `print()` line:

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

This invokes `performStoreRequest(with:)` 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 containing 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 Console:

```
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, your app has successfully talked 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 useful 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 should be something like this:

```
URL: 'https://itunes.apple.com/search?term=Metallica'
HTTP load failed (error code: -1009 [1:50]) for Task
<F5199AB7-5011-42FB-91B5-656244861482>.<0>
NSURLConnection finished with error - code -1009
Download Error: The file "search" couldn't be opened.
```

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.

Parse 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 *structured* data format. It typically consists of arrays and dictionaries that contain other arrays and dictionaries, as well as regular data such as strings and numbers.

An overview of the JSON data

The JSON from the iTunes store roughly looks like this:

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

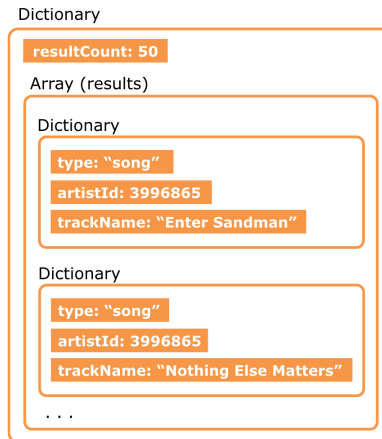
The { } brackets surround a dictionary. This particular dictionary has two keys: `resultCount` and `results`. The first one, `resultCount`, has a numeric value. This is the number of items that matched your search query. By default the limit is a maximum of 50 items, but as you shall later see, you can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

The `results` key contains an array, which is indicated 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 made up of 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 **JSON parser** you can turn this data into Swift Dictionary and Array objects.

JSON or XML?

JSON is not the only structured data format out there. XML, which stands for Extensible Markup Language, is a slightly more formal standard. Both formats serve the same purpose, but they look a bit different. If the iTunes store returned 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 certainly possible that if you want your app to talk to a particular web service, you'll be expected to deal with XML data.

Prepare to parse JSON data

In the past, if you wanted to parse JSON, it used to be necessary to include a third-party framework into your apps, or to manually walk through the data structure using the built-in iOS JSON parser. But with iOS 11, there's a new way to do things - your old pal `Codable`.

Remember how you used a `PropertyListDecoder` to decode plist data that supported the `Codable` protocol for reading (and saving) data in *Checklists*? Well, property lists aren't the only format supported by `Codable` out of the box - JSON is supported too!

All you need to do in order to allow your app to read JSON data directly into the relevant data structures is to set them up to conform to `Codable`!

"Now hold on there", I hear you saying. "How does `Codable` know how an arbitrary data structure from the Internet is set up in order to correctly extract the right bits of data?" Ah, it's all in how you set your data structures up. You'll understand as you proceed to parse the data you received from the iTunes server.

The whole trick to using `Codable` to parse JSON data is to set up your classes (or structs) to reflect the structure of the data that you'll parse. As you noticed above, there are two parts to the JSON response received from the iTunes server:

1. The response wrapper which contains the number of results and an array of results.
2. The array itself which is made up of individual search result items.

We need to model both of the above in order to parse the JSON data correctly. We've already made some headway in terms of modeling the search results by way of the `SearchResult` object, but we need to do some modifications in order to get the object ready for JSON parsing.

But first, let's add a new data model for the results wrapper.

► Open **`SearchResult.swift`** and replace its contents with the following:

```
class ResultArray:Codable {
    var resultCount = 0
    var results = [SearchResult]()
}

class SearchResult:Codable {
```

```
var artistName = ""  
var trackName = ""  
  
var name:String {  
    return trackName  
}  
}
```

There are a few changes here:

1. The `ResultArray` class models the response wrapper by containing a results count and an array of `SearchResult` objects. Note that this class supports the `Codable` protocol.

If you are wondering why this class is within the same file as `SearchResult`, it is simply for the sake of expediency. This class is not used anywhere else except as a temporary holder during the JSON parsing process. So I put it in the same file as `SearchResult`, which is the actual class you'll be using. But if you prefer, you can certainly put this class in a separate Swift file by itself. It does not make any difference to the app functionality.

2. The `SearchResult` class now supports the `Codable` protocol too. It also has a new property named `trackName` and the existing property for `name` has been converted to a computed property which currently returns the value of the `trackName` property.

The reason for the second set of changes (other than for `Codable` support) might not be obvious immediately. Take a look at the response data you received from the server. Did you notice the "kind" key?

The search results from iTunes can be for multiple types of items - songs, videos, movies, tv shows, books etc. That key indicates the type of item the search result is for. And depending on the item type, you might want to vary how you display an item name. For example, you might not always want to use the "trackName" key as the item name. The computed name property is simply preparation for the future in case you want to display different names depending on the result type.

Also, notice that now all the property names in the class match actual keys in the JSON data. (You can parse JSON even without the property names matching the key names, but that's a bit more complicated. So let's take the easy route here - baby steps ...)

And that's all you need in order to prepare for JSON parsing - onwards!

Parse the JSON data

You will be using the `JSONDecoder` class, appropriately enough, to parse JSON data. Only trouble is, `JSONDecoder` needs its input to be a `Data` object. You currently have the JSON response from the server as a `String`.

You can certainly convert the `String` to `Data` pretty easily, but it would be better to get the response from the server as `Data` in the first place. (You got the response from the server as `String` initially only to ensure that the response was correct.)

► Switch to **`SearchViewController.swift`** and modify `performStoreRequest(with:)` as follows:

```
func performStoreRequest(with url: URL) -> Data? {
    do {
        return try Data(contentsOf:url)    // Change this line
    } catch {
        . . .
    }
}
```

You simply change the request method to fetch the response from the server as `Data` instead of a `String`. The method now returns the value as an optional `Data` value since the fetch from the server can fail due to various reasons such as the Internet connection being down, the server being down etc.

► Add the following method to **`SearchViewController.swift`**:

```
func parse(data: Data) -> [SearchResult] {
    do {
        let decoder = JSONDecoder()
        let result = try decoder.decode(ResultArray.self, from:data)
        return result.results
    } catch {
        print("JSON Error: \(error)")
        return []
    }
}
```

You use a `JSONDecoder` object to convert the response data from the server to a temporary `ResultArray` object from which you extract the `results` property. Or at least, you *hope* you can convert the data without any issues...

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, `JSONDecoder` will not be able to parse the data and the app will return an empty array from `parse(data:)`.

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 is accessible via a different URL. Or, they might require 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.

In the case of the iTunes store web service, the top-level object *should* be a dictionary with two keys - one for the count, the other for the array of results - 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`. This in turn will cause `JSONDecoder` to fail parsing the data since it is no longer in the expected format.

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 `do-try-catch` block to check that the JSON parsing goes through fine. Should the conversion fail, then the app doesn’t burst into flames but simply returns an empty results array.

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.

► Modify `searchBarSearchButtonClicked(_)` as follows:

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
    if !searchBar.text!.isEmpty {
        . . .
        print("URL: \"\(url)\"")
        if let data = performStoreRequest(with: url) { // Modified
            let results = parse(data: data)           // New line
            print("Got results: \"\(results)\"")        // New line
        }
        tableView.reloadData()
    }
}
```

You simply change the constant for the result from the call to `performStoreRequest(with:)` (from `jsonString` to `data`), call the new `parse(data:)` method, and print its return value.

- Run the app and search for something. The Xcode Console now prints the following:

```
URL: 'https://itunes.apple.com/search?term=Metallica'
Got results: [StoreSearch.SearchResult, StoreSearch.SearchResult,
StoreSearch.SearchResult, StoreSearch.SearchResult,
StoreSearch.SearchResult, StoreSearch.SearchResult,
StoreSearch.SearchResult, StoreSearch.SearchResult,
StoreSearch.SearchResult, StoreSearch.SearchResult,
StoreSearch.SearchResult, StoreSearch.SearchResult,
StoreSearch.SearchResult, StoreSearch.SearchResult,
. . . ]
```

Hmm ... that certainly *looks* like an array of 50 items, but it doesn't really tell you anything much about the actual data - just that the array consists of `SearchResult` objects. That's not much good to you, is it?

Print object contents

- Modify the `SearchResult` class in **SearchResult.swift** to conform to the `CustomStringConvertible` protocol:

```
class SearchResult: Codable, CustomStringConvertible {
```

The `CustomStringConvertible` protocol allows an object to have a custom string representation. Or, to put it another way, the protocol allows objects to have a custom string describing the object (or its contents).

So, how does the protocol provide this string description? That is done via the protocol's `description` property.

- Add the following code to the `SearchResult` class:

```
var description: String {
    return "Name: \(name), Artist Name: \(artistName)"
}
```

The above is your implementation of the `description` property to conform to the `CustomStringConvertible`. For your `SearchResult` class, the description consists of the values of the `name` and `artistName` properties. You could output any string value from here but in this particular instance, those properties are probably the most appropriate since they help identify the object.

- Run the app again and search for something. The Xcode Console should now print something like the following:

```
URL: 'https://itunes.apple.com/search?term=Metallica'
Got results: [Name: Enter Sandman, Artist Name: Metallica, Name: Nothing
Else Matters, Artist Name: Metallica, Name: The Unforgiven, Artist Name:
```

```
Metallica, Name: One, Artist Name: Metallica, Name: Wherever I May Roam,  
Artist Name: Metallica,  
. . .
```

Yep, that looks more like it!

You have converted a bunch of JSON that didn't make a lot of sense, into actual objects that you can use.

Error handling

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

► Add the following method:

```
func showNetworkError() {  
    let alert = UIAlertController(title: "Whoops...",  
                                message: "There was an error accessing 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.

Note: The message variable is split into two separate strings and concatenated (or added together) using the plus (+) operator just so that the string would display nicely for this book. You can feel free to type out the whole string as a single string instead.

► Add the following line to `performStoreRequest(with:)` just before the `return nil:`

```
showNetworkError()
```

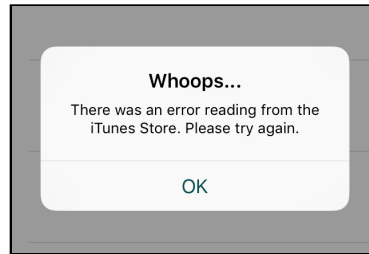
Simply put, if something goes wrong with the request to the iTunes store, you call `showNetworkError()` to show an alert box.

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 `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

It should be obvious that when you're doing networking, things can – and will! – go wrong, often in unexpected ways. So, it's always good to be prepared for surprises.

Work with the JSON results

So far you've managed to send a request to the iTunes web service and you parsed the JSON data into an array of `SearchResult` objects. However, we are not quite done.

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` field.

Let's modify our data model to load the value for the above key.

➤ Add the following properties to `SearchResult`:

```
var kind = ""
```

➤ Also modify the return line for `description` to:

```
return "Kind: \(kind), Name: \(name), Artist Name: \(artistName)\n"
```

➤ 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.

```
Kind: feature-movie, Name: Beaches, Artist Name: Garry Marshall  
Kind: song, Name: Wind Beneath My Wings, Artist Name: Bette Midler  
Kind: tv-episode, Name: Beaches, Artist Name: Dora the Explorer  
. . .
```

Now, let's 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 set up 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:

affiliate.itunes.apple.com/resources/documentation/itunes-store-web-service-search-api

Just reading the docs is often not enough though. 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. 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.

Load more properties

The current `SearchResult` class only has a few properties. 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 properties to **`SearchResult.swift`**:

```
var trackPrice = 0.0
var currency = ""
var artworkUrl60 = ""
var artworkUrl100 = ""
var trackViewUrl = ""
var primaryGenreName = ""
```

You're not including *everything* that the iTunes store returns, only the fields that are interesting to this app. Also, note that you've named the properties to match the keys in the JSON data exactly.

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

Provided the class supports `Codable`, with just the simple addition of new properties (as long as they are named the same as the JSON keys), you are now able to load these new values into your class.

But what if you don't want to use the not-quite-user-friendly names from the JSON data such as `artworkUrl60` or `artworkUrl100` but instead want to use more descriptive names such as `artworkSmall` and `artworkLarge`?

Never fear, `Codable` has support for that too :]

But before we get to that, you should run your app once to make sure that the above code changes didn't break anything. So, run your app, make a search, and verify that you still get output in the Xcode Console indicating that the search was successful.

All working fine? Great! Let's move on to naming the `SearchResults` properties to be as you want them and not as the JSON data sets them ...

Support better property names

► Replace the following lines of code in **SearchResult.swift**:

```
var artworkUrl60 = ""  
var artworkUrl100 = ""  
var trackViewUrl = ""  
var primaryGenreName = ""
```

With this:

```
var imageSmall = ""  
var imageLarge = ""  
var storeURL = ""  
var genre = ""  
  
enum CodingKeys: String, CodingKey {  
    case imageSmall = "artworkUrl60"  
    case imageLarge = "artworkUrl100"  
    case storeURL = "trackViewUrl"  
    case genre = "primaryGenreName"  
    case kind, artistName, trackName  
    case trackPrice, currency  
}
```

As you'll notice, you've changed the property names to be more descriptive, but what does the enum do?

As you've seen previously, an enum (or enumeration), is a way to have a list of values and names for those values. Here, you use the `CodingKeys` enumeration to let the `Codable` protocol know how you want the `SearchResult` properties matched to the JSON data.

Do note that if you do use the `CodingKeys` enumeration, it has to provide a case for all your properties in the class - the ones which map to a JSON key with the same name are the last two cases in the enum - you'll notice that they don't have a value specified.

That's all there is to it :) Run your app again (and maybe change the `description` property to return one of the new values to test they display correctly) and verify that the code still works with the new properties.

Use the results

With these latest changes, `searchBarSearchButtonClicked(_:)` retrieves an array of `SearchResult` objects populated with useful information, but you're not doing anything with that array yet.

► Switch to **SearchViewController.swift** and in `searchBarSearchButtonClicked(_:)`, replace the following lines:

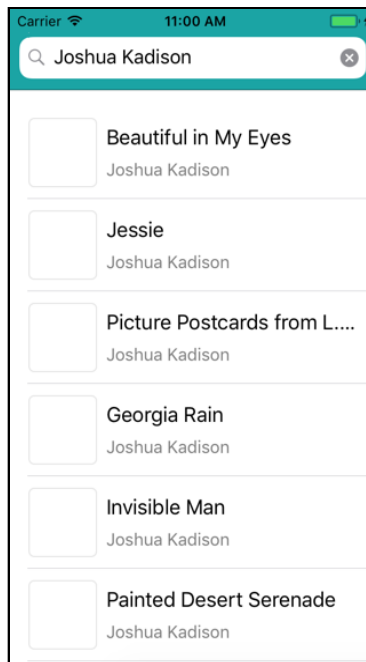
```
let results = parse(data: data)
print("Got results: \(results)")
```

With:

```
searchResults = parse(data: data)
```

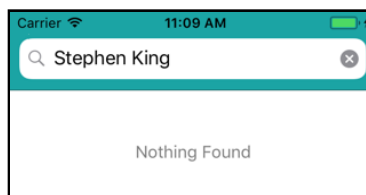
Instead of placing the results in a local variable and printing them out, you now place the returned array into the `searchResults` instance variable so that the table view can show the actual search result objects.

► 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!



The results from the search now show up in the table

That looks promising! How about a search for an author, for example, Stephen King?



There are no results for Stephen King

Huh? No results for Stephen King?? What gives?

If you check the Xcode Console, you'll notice a message similar to this:

```
"Key not found when expecting non-optional type String for coding key  
\"storeURL\"")
```

What does that mean?

Remember the `CodingKeys` enumeration? You specify that the `storeURL` property will get its value from a specific JSON key. But what if that key, `"trackViewUrl"`, was not there in the JSON data?

"Can that happen?" I hear you ask yourself. Of course, it can! Remember how I said that some items, such as audiobooks have different data structures? You have hit one such situation.

The biggest differences currently between the other item types and audiobooks is that audiobooks do not have certain JSON keys that are present for other items. Here's a breakdown:

1. **kind:** This value is not present at all.
2. **trackName:** Instead of `"trackName"`, you get `"collectionName"`.
3. **trackviewUrl:** Instead of this value, you have `"collectionViewUrl"` - which provides the iTunes link to the item.
4. **trackPrice:** Instead of `"trackPrice"`, you get `"collectionPrice"`.

Additionally, there are a few other JSON differences for a couple of item types:

1. Software and e-book items do not have `"trackPrice"` key, instead they have a `"price"` key.
2. E-books don't have a `"primaryGenreName"` key - they have an array of genres.

So how can you fix things so that the `JSONDecoder` can correctly decode the JSON data from the iTunes Store server no matter the type of item?

The `kind` property is the easiest one to handle, you mark it as an optional since it won't always have a value.

► In `SearchResult.swift`, change the `kind` property declaration to:

```
var kind:String?
```

► If you had the kind property being output for description, you will now get an Xcode warning about kind being an optional. So change the return to be something like this:

```
return "Kind: \(kind ?? ""), Name: \(name), Artist Name: \(artistName)\n"
```

Notice the ?? operator in the above line - it's called the *nil-coalescing operator*. The nil-coalescing operator unwraps the variable to the left of the operator if it has a value, if not, it returns the value to the right of the operator as the default value.

With those changes, you take care of the kind property. But what about the others, where sometimes a key can be present and sometimes not? All of those properties would have to be set as optionals as well. But that's only part of the solution. Remember how you added a computed variable called name which returns the trackName? This is where that comes into play ...

If you add another variable to store collectionName - the name of the item when it is an audiobook - then you can return the correct value from name depending on the case. You can do something similar for the store URL and price as well.

Let's make the necessary changes.

► Make trackName and trackPrice optionals:

```
var trackName:String?  
var trackPrice:Double?
```

► Remove the storeURL property from SearchResult - you'll add two separate optional properties for the audiobook and non-audiobook types. Also remove the storeURL case from CodingKeys.

► Remove the genre property from SearchResult - you'll add two separate optional properties for the e-book and non-e-book types. Also remove the genre case from CodingKeys.

► Add new optional properties for the variant keys present in the special items mentioned above:

```
var trackViewUrl:String?  
var collectionName:String?  
var collectionViewUrl:String?  
var collectionPrice:Double?  
var itemPrice:Double?  
var itemGenre:String?  
var bookGenre:[String]?
```

► Replace the name computed property with the following:

```
var name:String {  
    return trackName ?? collectionName ?? ""  
}
```

The change is simple enough, except for the *chaining* of the nil-coalescing operator. You check to see if trackName is nil - if not, you return the unwrapped value of trackName. If trackName is nil, you move on to collectionName and do the same check. If both values are nil, you return an empty string.

► Add the following three new computed properties:

```
var storeURL:String {  
    return trackViewUrl ?? collectionViewUrl ?? ""  
}  
  
var price:Double {  
    return trackPrice ?? collectionPrice ?? itemPrice ?? 0.0  
}  
  
var genre:String {  
    if let genre = itemGenre {  
        return genre  
    } else if let genres = bookGenre {  
        return genres.joined(separator: ", ")  
    }  
    return ""  
}
```

The first two computed properties work similar to how the name computed property works. So nothing new there. The genre property simply returns the genre for items which are not e-books. For e-books, the method combines all the genre values in the array separated by commas and then returns the combined string.

All that remains is to add all the new properties to the CodingKeys enumeration - if you don't, some of the values might not be populated correctly during JSON decoding. Once you're done, CodingKeys should look like this:

```
enum CodingKeys: String, CodingKey {  
    case imageSmall = "artworkUrl60"  
    case imageLarge = "artworkUrl100"  
    case itemGenre = "primaryGenreName"  
    case bookGenre = "genres"  
    case itemPrice = "price"  
    case kind, artistName, currency  
    case trackName, trackPrice, trackViewUrl  
    case collectionName, collectionViewUrl, collectionPrice  
}
```

► Run the app again, search for "Stephen King" and you should get some results for the master of horror this time!

Show the product type

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.

► Still in **SearchResult.swift**, add the following computed property:

```
var type:String {  
    return kind ?? "audiobook"  
}
```

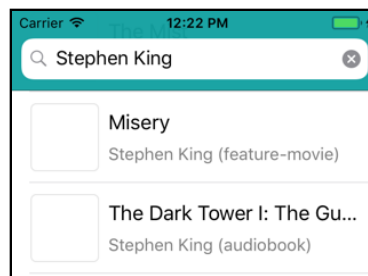
Remember that `kind` could be `nil` if the item type is an audiobook. You hedge against that with this new computed property.

► Open **SearchViewController.swift** and 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.type)  
}
```

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 new `type` 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 for *StoreSearch*). It’s better to convert this internal identifier (“feature-movie”) into the text that you want to show to the user (“Movie”).

► Replace the type computed property in **SearchResult.swift** with this one:

```
var type:String {
    let kind = self.kind ?? "audiobook"
    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: break
    }
    return "Unknown"
}
```

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 a string saying “Unknown” (and hopefully fix the unknown type in an update of the app).

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 `kind`. Swift needs to know what to do when `kind` is not any of the known values. That’s why you’re required to include the `default: case`, as a catchall for any other possible values of `kind`.

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.

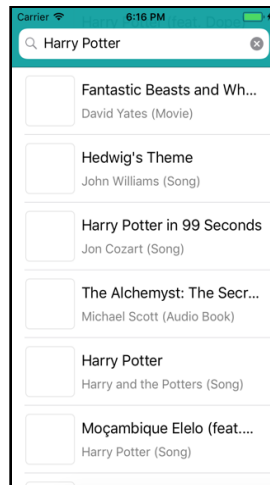
Now the item type should display not as a value from the web service, but instead, as the value you set for each item type:



The product type is a bit more human-friendly

► 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 on, 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

Sort the search results

It'd be nice to sort the search results alphabetically. That's actually quite easy. A Swift Array already has a method to sort itself. All you have to do is tell it what to sort on.

► In **SearchViewController.swift**, in `searchBarSearchButtonClicked(_:)`, right after the call to `parse(data:)` add the following:

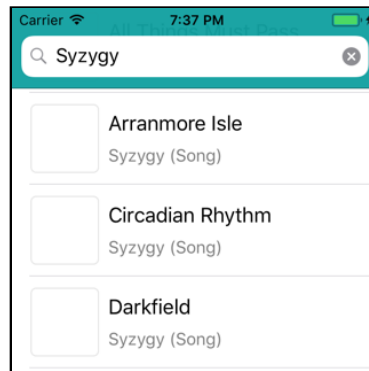
```
searchResults.sort(by: { result1, result2 in
    return result1.name.localizedStandardCompare(
        result2.name) == .orderedAscending
    })
```

After the results array is fetched, you call `sort(by:)` on the `searchResults` array with a closure that determines the sorting rules. This is identical to what you did in *Checklists* 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 comparison of the two objects uses `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.

Improve the sorting code

► Change the sorting code you just added to:

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

This uses the *trailing* closure syntax to put the closure after 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 no longer refer to the two `SearchResult` objects by name but as the special `$0` and `$1` variables. Using this shorthand 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 `+` or `*` 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 for 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 with a message such as "Add fetching data from web service using synchronous network request".

You can find the project files for this chapter under **34 – Networking** in the Source Code folder.

Chapter 35: Asynchronous Networking

You've got your app doing network searches and it is working well. The synchronous network calls aren't so bad, are they?

Yes they are, and I'll show you why! Did you notice that whenever you performed a search, the app became unresponsive? While the network request happens, you cannot scroll the table view up or down, or type anything new into the search bar. The app is completely frozen for a few seconds.

You may not have seen this if your network connection is 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.

To most users, an app that does not respond is an app that has crashed. The user will probably 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.

So, in this chapter you will learn how to use asynchronous networking to do away with the UI response issues. You'll do the following:

- **Extreme synchronous networking:** Learn how synchronous networking can affect the performance of your app by dialing up the synchronous networking to the maximum.
- **The activity indicator:** Add an activity indicator to show when a search is going on so that the user knows something is happening.
- **Make it asynchronous:** Change the code to run web service requests to run on a background thread so that it does not lock up the app.

Extreme synchronous networking

Still not convinced of the evils of synchronous networking? 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 the app receives - by adding a "limit" parameter to the URL, you can set the maximum number of results that the web service will return. The default value is 50, the maximum is 200.

► Open **SearchViewController.swift** and in `iTunesURL(searchText:)`, change the line with the web service URL to the following:

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

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

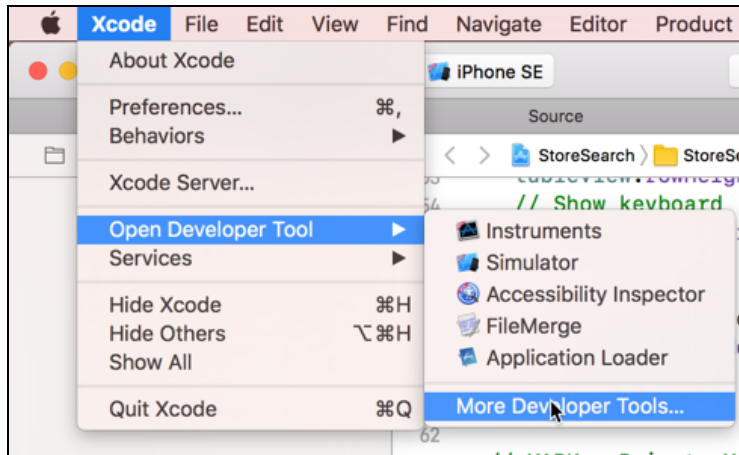
► If you run the app now, the search should be quite a bit slower.

The network link conditioner

Still too fast for you to see any app response issues? Then use the **Network Link Conditioner**. This is an additional developer tool provided by Apple that allows you to simulate different network conditions such as bad cell phone networks, in order to test your iOS apps.

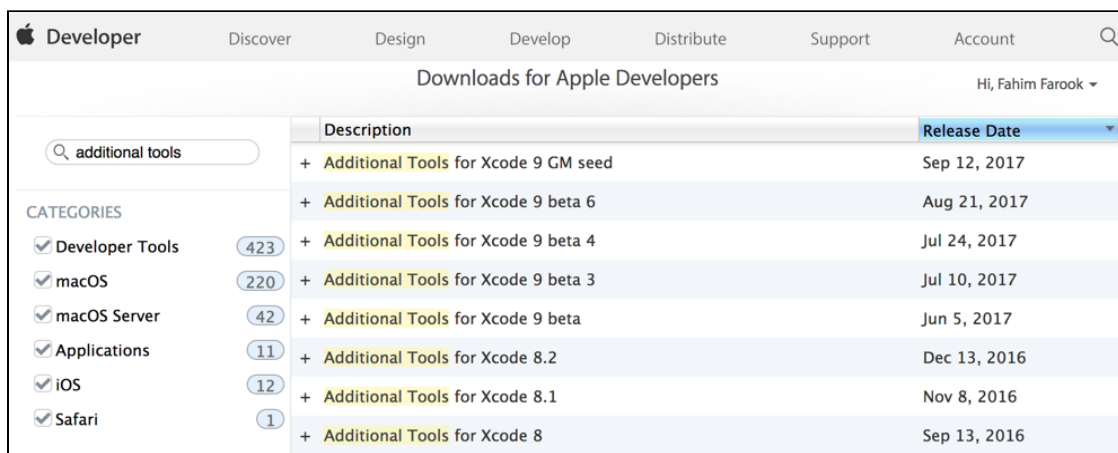
But first, before you can use it, you probably have to install the Network Link Conditioner since that is not something that is installed by default either as part of macOS or as part of your Xcode installation.

► Select **Open Developer Tool → More Developer Tools...** from the Xcode menu.



The More Developer Tools menu option

This should open the [Downloads for Apple Developers](#) web page in your default browser. (You might be asked to login to the Apple Developer portal first since this is a resource which is only available to registered Apple developers.)

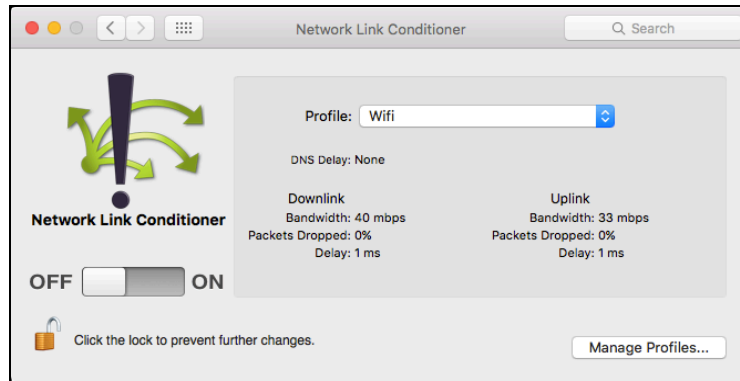


The Downloads for Apple Developers page

► As shown in the screenshot, search for "additional tools". You should get a list of different downloads. Select the most recent one (based on Release Date), download it, open the DMG file, switch to the **Hardware** folder on the DMG, and then double-click **Network Link Conditioner.prefPane** to install it.

The Network Link Conditioner is now available to you to use as a System Preferences panel option.

► Open **System Preferences** on your Mac and locate **Network Link Conditioner** (it should be in the last section of items, at the very bottom).

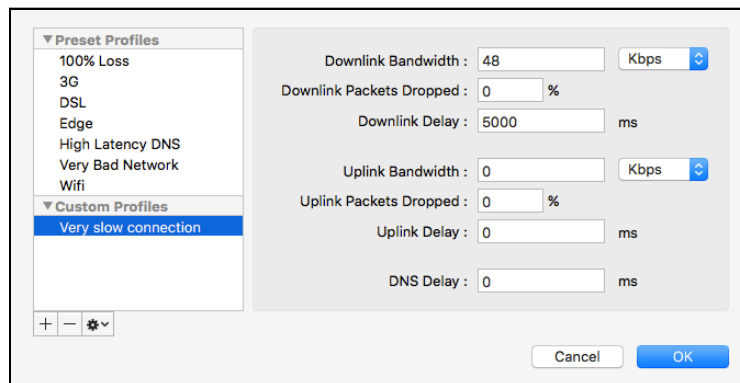


The Network Link Conditioner preference pane

Let's simulate a really slow connection.

► Click on **Manage Profiles** and create a new profile (by clicking the plus button on the bottom left) 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 the Network Link Conditioner.

► 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 force kill it, in which case it really does crash. You don't want that to happen!

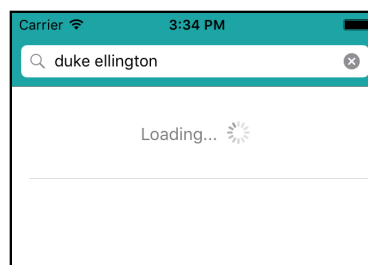
"Ah," you say, "let's show some type of animation to let the user know that the app is communicating with a server. Then at least they will know that the app is busy."

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 a 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:

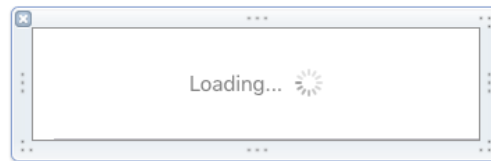


The app shows that it is busy

The activity indicator table view cell

- Create a new, empty nib file. Call it **LoadingCell.xib**.
- Drag a new **Table View Cell** on to 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 should look like this:



The design of the LoadingCell nib

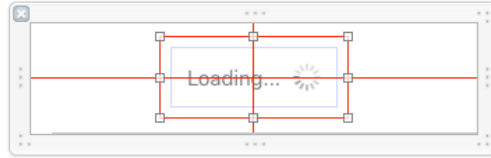
To make this cell work properly on larger screens, you'll add constraints that keep the label and the activity spinner centered in the cell. The easiest way to do this is to place these two items inside a container view and center that.

- 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 the selected views.



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.

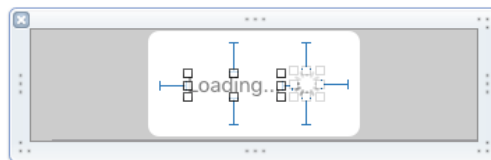
To fix this, you're going to add constraints to the label and activity indicator as well, so that the width and height of the container view are determined by the size of the two things inside it.

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 **Add New Constraints** button. Simply pin it to all four sides 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 screen.

Use the activity indicator cell

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 `TableViewCellIdentifiers` structure 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)
```

You now 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 . . .
```

The `if isLoading` condition returns 1, because you need a row in order to show a cell.

- Update `tableView(_:cellForRowAt:)` as follows:

```
func tableView(_ tableView: UITableView,
               cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    // New code
    if isLoading {
        let cell = tableView.dequeueReusableCell(withIdentifier:
            TableViewCellIdentifiers.loadingCell, for: indexPath)

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

You added an if condition 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 {    // Changed
        return nil
    } else {
        return indexPath
    }
}
```

You added `|| isLoading` to the if statement. Just like you don't want 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.

There's only one thing remaining: 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()
        // New code
        isLoading = true
        tableView.reloadData()
        // End of new code
        . . .
        isLoading = false                // New code
        tableView.reloadData()
    }
}
```

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!

Test the new loading cell

► 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(_:)` as follows:

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
    if !searchBar.text!.isEmpty {
        searchBar.resignFirstResponder()
        isLoading = true
        tableView.reloadData()
        /*
        . . . the networking code (commented out) . . .
        */
    }
}
```

Note that you don't have to remove anything from the code - simply comment out everything after the first call to `tableView.reloadData()`.

► 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 is not 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 a 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.

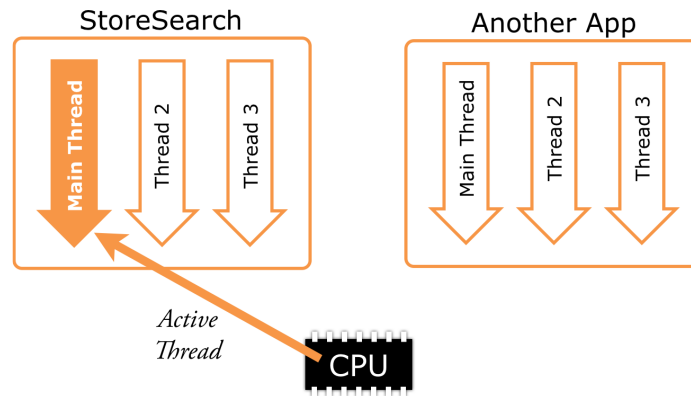
With so few cores available, how come modern computers can have many more applications and other processes running at the same time? (I count 287 active processes on my Mac right now.)

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.

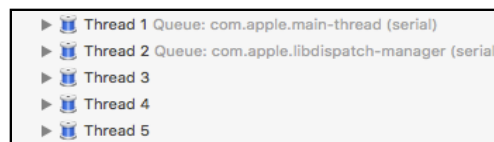
Each process contains one or more **threads**. I just mentioned that each process in turn is given a bit of CPU time to do its work. The process splits up that time among its threads. Each thread typically performs its own work and is as independent as possible from the other threads within that process.

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 because that would lock up your main thread.

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 for a long time.

This is why the current synchronous approach to doing networking is bad: **Never block the main thread**. It's one of the cardinal sins of iOS programming!

Make it asynchronous

To prevent blocking the main thread, any operation that might take a while to complete should be **asynchronous**. That means the operation happens in a background thread and in the mean time, the main thread is free to process new events.

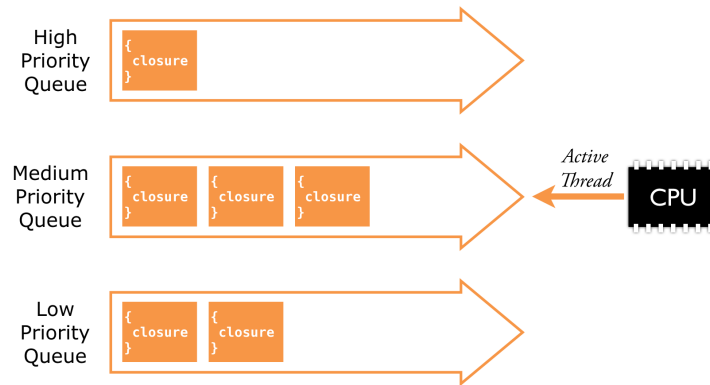
That is not to say you should create your own thread. If you've programmed on other platforms before, you may not think twice about creating new threads, but on iOS that is often not the best solution.

You see, threads are tricky. Not threads per se, but doing things in parallel. 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 *MyLocations*, but now you'll put it to even better use.

In short, GCD has a number of queues with different priorities. To perform a job in the background, you put the job in a closure and then pass that closure to a queue and forget about it. It's as simple as that.

GCD will get 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 job.



Queues have a list of closures to perform on a background thread

Put the web request in 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(_:)` as follows:

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
    if !searchBar.text!.isEmpty {
        . . .
        searchResults = []
        // Replace all code after this with new code below
        // 1
        let queue = DispatchQueue.global()
        // 2
        queue.async {
            let url = self.iTunesURL(searchText: searchBar.text!)

            if let data = self.performStoreRequest(with: url) {
                self.searchResults = self.parse(data: data)
                self.searchResults.sort(by: <)
                // 3
                print("DONE!")
                return
            }
        }
    }
}
```

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 - everything between `queue.async {` and the closing `}` is the closure. Whatever code 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 Console. (You might also see something which looks like an error about a UI API being called in a background thread - don't worry about that yet, we'll get to that soon...)

Of course, the Loading... cell sticks around forever because you still haven't told it to go away.

Put UI updates on the main thread

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 on a background thread, not the main thread.

As it happens, there is also a "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 in `searchBarSearchButtonClicked(_:)` 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.

► 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!

But ... there's still that pesky error (or warning maybe?) about a UI API being called on a background thread - what's that about?

All kinds of queues

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  
    }  
}
```

Basically, while you do your work in a background thread, you still have to switch over to the main thread to do any user interface updates. That's just the way it is.

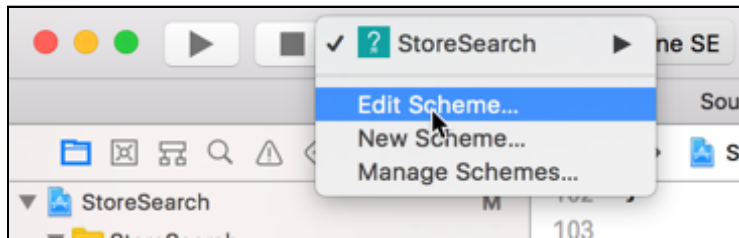
There is also `queue.sync`, without the “a”, which takes the next closure from the queue and performs it in the background, but makes you wait until that closure is done. That can be useful in some cases but most of the time you'll want to use `queue.async`. No one likes to wait!

The main thread checker

I mentioned previously that you should not run UI code on a background thread. However, till iOS 11, there was no easy way to discover UI code running on background threads except by scouring the source code laboriously line-by-line trying to determine what code ran on the main thread and what ran on a background thread.

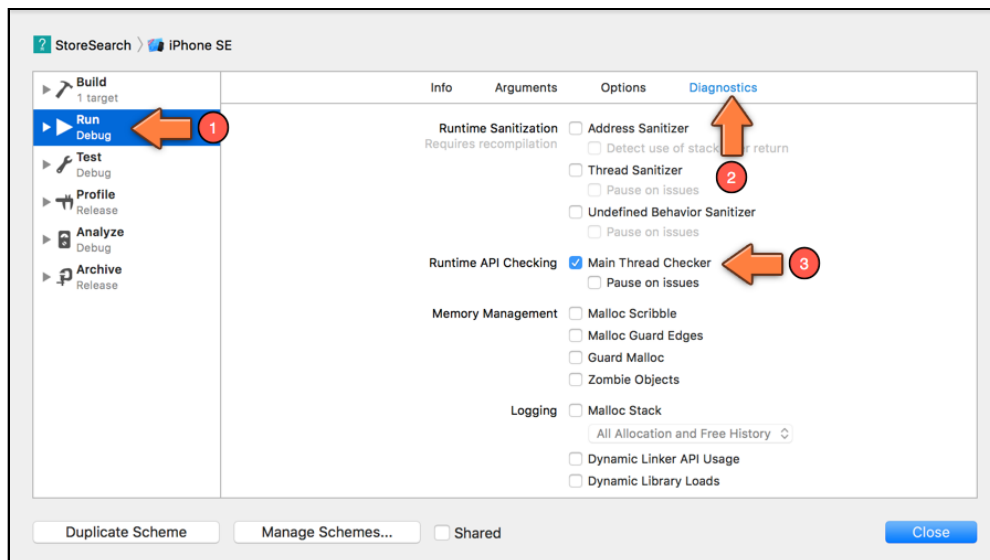
With iOS 11, Apple introduced a new diagnostic setting called the Main Thread Checker which would warn you if you had any UI code running on a background thread. This setting is supposed to be enabled by default, but if it is not, you can enable it quite easily - and I would suggest that you have it enabled at all times if possible since it can be quite invaluable.

- Click on the scheme dropdown in the Xcode toolbar and select Edit Scheme...



Edit scheme

- Select **Run** in the left panel, switch to the **Diagnostics** tab, and make sure **Main Thread Checker** is checked under Runtime API Checking.



Main Thread Checker setting

If you didn't have the Main Thread Checker enabled before, it is now enabled. With the setting enabled, if you run StoreSearch and do a search for an item, you should see something like the following in the Xcode Console:

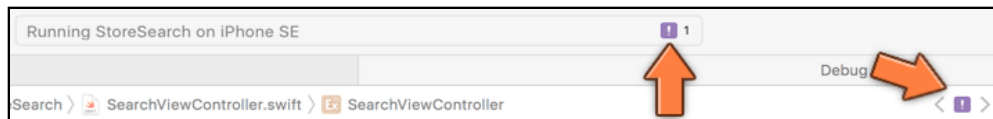
```
[reports] Main Thread Checker: UI API called on a background thread: -
[UISearchBar text]
PID: 39697, TID: 8913330, Thread name: (none), Queue name:
com.apple.root.default-qos, QoS: 21
Backtrace:
4   StoreSearch                                0x000000010adbcb98
```

```

_T011StoreSearch0B14ViewController09searchBarB13ButtonClickeddySo08UISear
chF0CFyyU_ + 104
5  StoreSearch                                0x0000000010adbd620
_T011StoreSearch0B14ViewController09searchBarB13ButtonClickeddySo08UISear
chF0CFyyU_TA + 80
6  StoreSearch                                0x0000000010adbbf19 _T0Ix_IyB_TR +
41
7  libdispatch.dylib                          0x000000001100a0711
_dispatch_call_block_and_release + 12
8  libdispatch.dylib                          0x000000001100a17a0
_dispatch_client_callout + 8
9  libdispatch.dylib                          0x000000001100a6834
_dispatch_queue_override_invoke + 1507
10 libdispatch.dylib                          0x000000001100add79
_dispatch_root_queue_drain + 785
11 libdispatch.dylib                          0x000000001100ada05
_dispatch_worker_thread4 + 54
12 libsystem_pthread.dylib                    0x000000001105115a2
_pthread_wqthread + 1299
13 libsystem_pthread.dylib                    0x0000000011051107d start_wqthread
+ 13

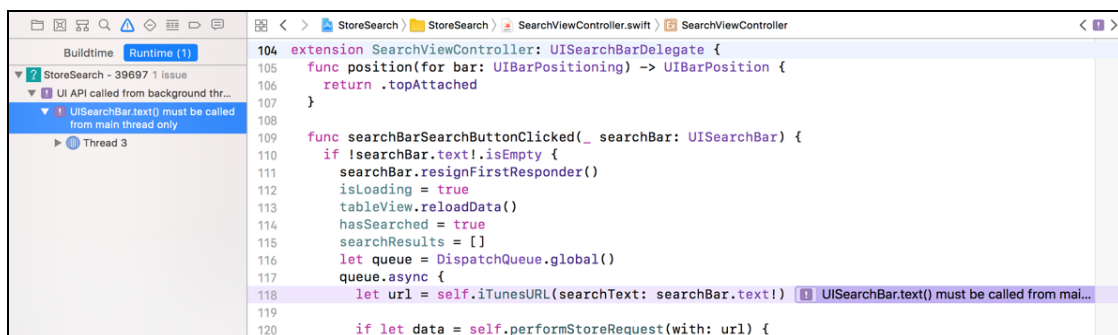
```

You might also notice that the Xcode toolbar's activity view now has a purple icon and that there's a purple icon on the right corner of the jump bar, where errors are normally displayed.



Purple icons indicating Main Thread Checker issues

If you click on the icon in the activity view, you will be taken to the **Runtime** tab of the **Issue navigator**, where you can click on the issue to be taken to the offending line in your source code:



Issue navigator

And you finally see what the issue is - you access the data from a UI control, the Search Bar, in a background thread. It might be better to do this in the main thread. The fix is simple, move this line:

```
let url = self.iTunesURL(searchText: searchBar.text!)
```

Above the `queue.async {` line. That is all you need to do, the code will still work fine.

Don't believe me? Fine, try running the code after you've made the change, do a search and see if you get the Main Thread Checker error. Convinced? :]

Commit your code

► 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 will have to do this as two separate steps - first create a commit with a suitable message, and then create a tag for your latest commit.

You can find the project files for this chapter under **35 – Asynchronous Networking** in the Source Code folder.

Chapter 36: URLSession

So far, you've used the `Data(contentsOf:)` 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 chapter 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.

You'll cover the following items in this chapter:

- **Branch it:** Creating Git branches for major code changes.
- **Put `URLSession` into action:** Use the `URLSession` class for asynchronous networking instead of downloading the contents of a URL directly.
- **Cancel operations:** Canceling a running network request when a second network request is initiated.
- **Search different categories:** Allow the user to select a specific iTunes Store category to search in instead of returning items from all categories.
- **Download the artwork:** Download the images for search result items and display them as part of the search result listing.
- **Merge the branch:** Merge your changes from your working Git branch back to your master branch.

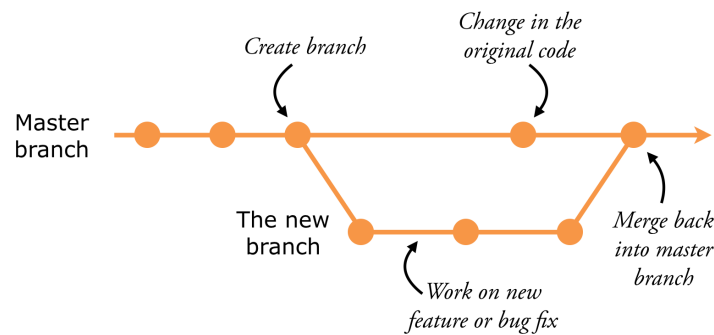
Branch it

Whenever you make a big change to the code, such as replacing all the networking stuff with `URLSession`, there is a possibility that you'll mess things up. I certainly do often enough! That's why it's smart to create a 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 `URLSession` 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.

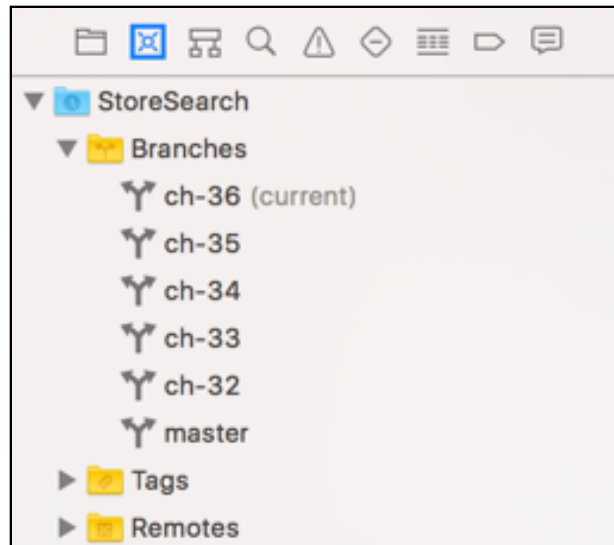


Branches in action

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.

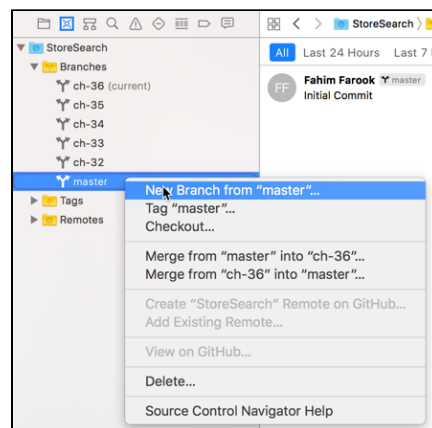
You can find the branches for your repository in the **Source Control** navigator:



The Source Control branch list

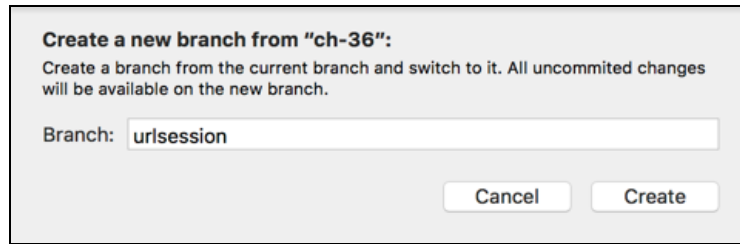
Note: In my case, for the above screenshot, I have multiple branches already - I have a branch for each chapter of the book. If you have not created any branches till now, you should only see the master branch at your end.

► Select **master** (or whatever is your current branch) from the branch list, and right-click on the branch name to get a context-menu with possible actions. Select **New Branch from "master"...**:



The branch context-menu

► You will get a new dialog asking for the new branch name. Enter **urlsession** as the new 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 now the current one.

This new branch contains the exact same source code and history as the master branch (or whichever branch you used as the parent for the new branch). But from here on out the two paths will diverge – any changes you make happen on the "urlsession" branch only.

Put URLSession into action

Good, now that you're in a new branch, it's safe to experiment with these new APIs.

► First, remove `performStoreRequest(with:)` from **SearchViewController.swift**. Yup, that's right, you won't be needing that method 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 "original" one.

Anyway, on to `URLSession`. This is a closed-based API, meaning that instead of making a delegate, you pass 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(_:)` as follows:

```
func searchBarSearchButtonClicked(_ searchBar: UISearchBar) {
    if !searchBar.text!.isEmpty {
        . . .
        searchResults = []
        // Replace all code after this with new code below
        // 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 using the search text, just like before.
2. Get a shared URLSession instance, which uses the 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 fetching the contents of a given URL. The code from the completion handler will be invoked when the data task has received a response 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 there is 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 data fetched 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 on a background thread. So, the app is immediately free to continue (URLSession is as asynchronous as they come).

With these 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 a Console message saying “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. Instead, 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 holds on to 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 code.

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 appears after 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 `dataFormatter` 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: myHandler)
. . .
func myHandler(data: Data?, response: URLResponse?,
               error: Error?) {
    . . .
}
```

The above 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 a property 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 property. 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 URLSession 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. As you add more functionality to

StoreSearch you *will* have to use `[weak self]` with `URLSession` 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!

Handle status codes

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](http://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 the received 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 - you'll consider the job a success only if it is 200.

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! 295831 bytes
```

Since your received data is in the form of a `Data` object, unlike text, its content can't be printed out. So, you just get the length of the data instead.

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 URL 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 searchLOL page – and the app correctly considers this a failure. That’s a good thing too, because (if you were to convert the value of data to text) 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 /searchLOL 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 parse received JSON data.

Parse the data

- First, put `iTunesURL(searchText:)` back to the way it was (⌘+Z to undo).
- In the completion handler, replace the `print("Success! \(data)")` line with:

```
if let data = data {
    self.searchResults = self.parse(data: data)
    self.searchResults.sort(by: <)
    DispatchQueue.main.async {
        self.isLoading = false
        self.tableView.reloadData()
    }
    return
}
```

This unwraps the optional object from the data parameter and then calls `parse(data:)` 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 rule. That's why you wrap the reloading of the table view in a `DispatchQueue.main.async` closure.

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: If you want to determine via code whether a particular piece of code is being run on the main thread or not, 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.

Handle errors

► At the very end 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 execution reaches here only 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 successfully loading the data, 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 for 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.

Cancel operations

What happens when a search takes a long time and the user starts a second search while the first one is still going? The app doesn't disable the search bar, so it's possible for the user to do this. 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. They might see the results from their first search, which they are no longer expecting, only for that to be replaced by the results of the second search a few seconds later. Confusing!

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 will sit still and wait until the request is done.

You can usually fix this 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 initiates a new request.

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. URLSession makes this easy: data tasks have a `cancel()` method.

When you created the data task, you were given a `URLSessionDataTask` object, and you placed this into a local constant named `dataTask`. Cancelling the task, however, needs to happen the *next* time `searchBarSearchButtonClicked(_:)` is called.

Storing the `URLSessionDataTask` object into a local variable isn't good enough anymore; you need to keep that reference beyond the scope of the method. In other words, you have to store it in 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.

➤ In `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 end 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 is 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 has been 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 notice 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={NSErrorFailingURLStringKey=https://itunes.apple.com/search?term=Jade&limit=200, NSLocalizedDescription=cancelled, NSErrorFailingURLKey=https://itunes.apple.com/search?term=Jade&limit=200}
```

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 completion handler, change the `if let error` section to:

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

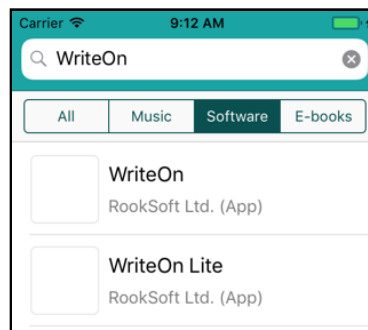
This simply ends the closure when there is an error with code -999. The rest of the closure gets skipped.

► 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, it is a good time to commit.

Search 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 will look 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 a set of choices.

Add the segmented control

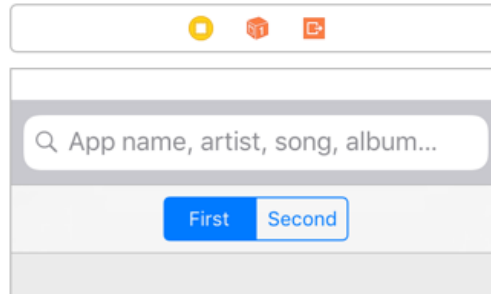
► Open the storyboard. Drag a new **Navigation Bar** into the view and put it below the Search Bar. You're using the Navigation Bar purely for decorative purposes, as a container for the segmented control.

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 Document Outline and drop it below the Search Bar. Then change its Y-position to 76.

► With the Navigation Bar selected, open the **Add New Constraints menu** and pin its **top**, **left**, and **right** sides.

- Drag a new **Segmented Control** from the Object Library on to the Navigation Bar's title (so it will replace the title).

The design should now look like this:

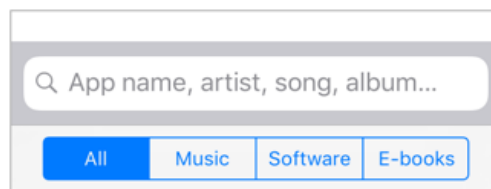


The Segmented Control sits in a Navigation Bar below the Search Bar

- 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 by changing the **Segment** dropdown in the Attributes inspector to select the correct segment.

The scene should look like this now:



The finished Segmented Control

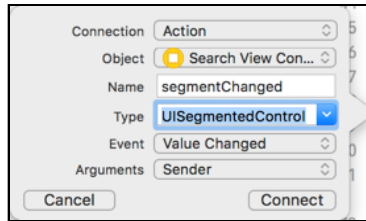
Next, you'll add a new outlet and action method for the Segmented Control. This is a good opportunity to practice using the Assistant editor.

Use the assistant editor

- Press **Option+⌘+Enter** to open the Assistant editor and then Control-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. Control-drag from the Segmented Control into the source code again, but this time choose:



Adding an action method for the segmented control

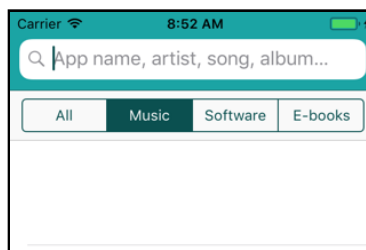
- Connection: **Action**
- Name: **segmentChanged**
- Type: **UISegmentedControl**
- Event: Value Changed
- Arguments: Sender

► Press **Connect** to add the action method. Then, add a `print()` statement to the new method:

```
@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 Console.



The segmented control in action

Use the segmented control

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 the new product category.

That means a search can now be triggered by two different events: tapping the Search button on the keyboard and selecting an item in the Segmented Control.

- Rename the `searchBarSearchButtonClicked(_:)` method to `performSearch()` and 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` property with that name and any references to `searchBar` in `performSearch()` will simply use that property.

- Now add a new version of `searchBarSearchButtonClicked(_:)` to 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.)

There is one thing left to be done - you have to tell the app to use the category based on 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 kind: String
    switch category {
        case 1: kind = "musicTrack"
        case 2: kind = "software"
        case 3: kind = "ebook"
        default: kind = ""
    }

    let encodedText = searchText.addingPercentEncoding(
        withAllowedCharacters: CharacterSet.urlQueryAllowed)!

    let urlString = "https://itunes.apple.com/search?" +
        "term=\(encodedText)&limit=200&entity=\(kind)"

    let url = URL(string: urlString)
    return url!
}
```

This first turns the category index from a number into a string, `kind`. (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. Also note that instead of calling `String(format:)`, you now construct the URL string using string interpolation.

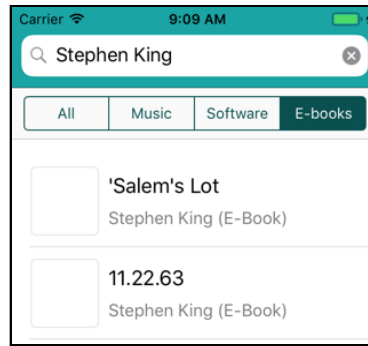
► In `performSearch()`, change the line that gets the URL to the following:

```
let url = 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 songs to movies to podcasts to audio books. But if all you wanted were to get to his books, you can now use the E-Books category to 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.

Set the launch screen

- 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, 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 models.

- Commit the changes and get ready for some more networking!

Download the artwork

The JSON search results contain a number of URLs to images and you put two of those – `imageSmall` and `imageLarge` – into the `SearchResult` object. Now you are going to download these images over the Internet and display them in the table view cells.

Downloading images, just like using a web service, is simply a matter of doing an HTTP request to a server that is connected to the Internet. An example of such a URL is:

<http://is4.mzstatic.com/image/thumb/Video1/v4/9c/d2/a5/9cd2a5e5-4710-abf4-925f-377e1666b0de/source/100x100bb.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 `is4.mzstatic.com`, but that doesn't matter at all 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 what kind of file it is.

There are various ways that you can download files from the Internet. You're going to use `URLSession` and write a handy `UIImageView` extension to make this really convenient. Of course, you'll be downloading these images asynchronously!

SearchResultCell refactoring

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`**:

```
// MARK:- Public Methods
func configure(for result: SearchResult) {
    nameLabel.text = result.name

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

This is the same code as in `tableView(_:cellForRowAt:)`.

➤ Now, change `tableView(_:cellForRowAt:)` as follows:

```
func tableView(_ tableView: UITableView,
               cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    if isLoading {
        . . .
    } else if searchResults.count == 0 {
        . . .
    } else {
        . . .
        let searchResult = searchResults[indexPath.row]
        // Replace all code after this with new code below
        cell.configure(for: searchResult)
        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 your last Git commit.)

➤ Run the app to make sure everything still works fine.

UIImageView extension for downloading images

OK, here comes the cool part. You will now add an extension for `UIImageView` that downloads the image and automatically displays it via 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 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 the 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 weakSelf = self {
                            weakSelf.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 create an image from that. It's possible that constructing the UIImage fails, for example, 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 might have navigated away to a different part of the app by then.

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 on 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 if necessary. 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.

Use the image downloader extension

► Switch to **SearchResultCell.swift** and add a new instance variable, `downloadTask`, to hold a reference to the image downloader:

```
var downloadTask: URLSessionDownloadTask?
```

► Now, add the following lines to the end of `configure(for:)`:

```
artworkImageView.image = UIImage(named: "Placeholder")
if let smallURL = URL(string: result.imageSmall) {
    downloadTask = artworkImageView.loadImage(url: smallURL)
}
```

This tells the `UIImageView` to load the image from `imageSmall` 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).

► Run the app and look at that... error message?!

The Xcode console should show something like this (alongwith a ton of error messages for failed download tasks):

```
App Transport Security has blocked a cleartext HTTP (http://) resource
load since it is insecure. Temporary exceptions can be configured via
your app's Info.plist file.
```

Override app transport security

As of iOS 9, you can no longer download files over HTTP. Instead, you always need to use HTTPS. 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 is not configured for HTTPS. So, `URLSession` cannot use a secure connection and therefore the request fails.

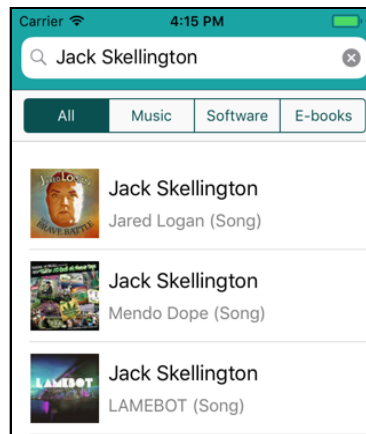
Fortunately, as the error message indicates, you can add a key to the app's Info.plist to bypass this App Transport Security feature, allowing you to use plain `http://` URLs.

- 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.

▼ App Transport Security Settings	◇	Dictionary	(1 item)
Allow Arbitrary Loads	◇	Boolean	YES

Overriding App Transport Security

- Run the app and look at those images!



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 support 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`. Under that dictionary, you can 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:

▼ App Transport Security Settings	↕	Dictionary	(2 items)
Allow Arbitrary Loads	↕	Boolean	NO
▼ Exception Domains	↕	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 might 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!

Cancel previous image downloads

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 downloading.

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
}
```

You simply 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 it happen, 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.

Caching

Depending on what you searched for, you may have noticed that many of the images were the same. For example, you might get 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.

Images aren't the only things 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 apps, 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 - the cached data only stays in the computer's working memory. But it is also possible to cache the data to 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 the documentation for `URLCache` and `URLSessionConfiguration` to learn more.

Merge the branch

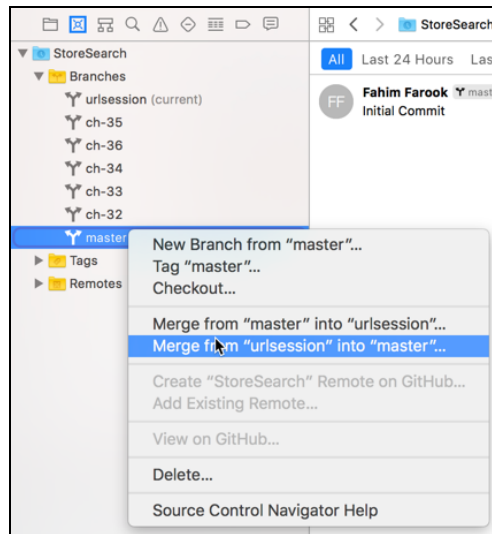
This concludes the section on talking to the web service and downloading images. Later on, you'll tweak the web service requests a bit more to include the user's language and country, but for now, you're done with this feature. I hope you got a good glimpse of what is possible with web services and how easy it is to build this functionality into your apps using `URLSession`.

➤ Commit these latest changes to the repository.

Merge the branch using Xcode

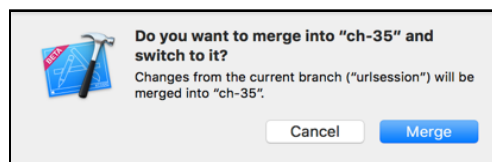
Now that you've completed a feature, you can merge this temporary branch back into the master branch.

- Switch to the **Source Control navigator**, select the **master** branch under branches, and right-click to get the context menu of actions. Select **Merge from "urlsession" into "master"...**:



Merging your changes back to the master branch

- You'll get a confirmation dialog. Click **Merge** if you want to continue.



The confirmation dialog before merging changes

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.

Merge the branch from the command line

Some of the source control features in Xcode are still a bit rough around the edges. So, it is possible that certain command, especially merging changes, might not work correctly. If Xcode didn't want to cooperate when you tried to merge changes, 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 saves any uncommitted changes so you can later restore them, if need be.

```
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 it. Xcode's Git support has improved a lot with Xcode 9, but for more complex things like merges you might be better off using the command line. It's well worth learning!

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.

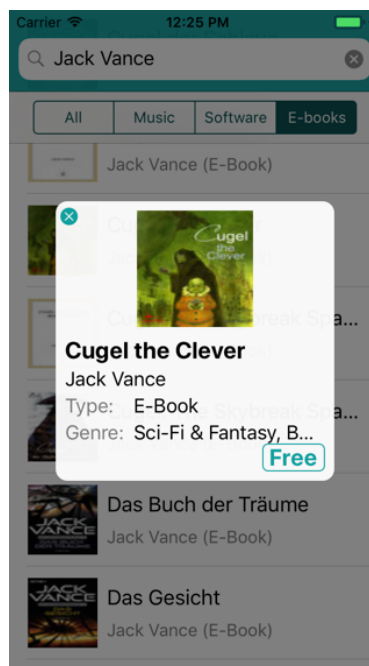
The most popular library at this point is AFNetworking, written in Objective-C but very usable from Swift (github.com/AFNetworking). A native Swift library is Alamofire (github.com/Alamofire).

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.

You can find the project files for this chapter under **36 – URLSession** in the Source Code folder.

Chapter 37: 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!

This chapter will cover the following:

- **The new view controller:** Create the bare minimum necessary for the new Detail pop-up and add the code to show/hide the pop-up.
- **Ad the rest of the controls:** Complete the design for the Detail pop-up.
- **Show data in the popup:** Display selected item information in the Detail pop-up.

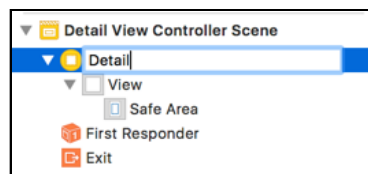
The new view controller

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

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.

The basic view controller

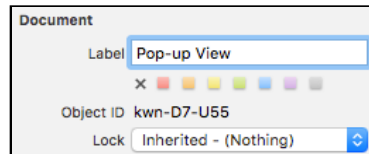
- Add a new **Cocoa Touch Class** file to the project. Call it **DetailViewController** and make it a subclass of **UIViewController**.
- Open the storyboard and drag a new **View Controller** on to the canvas. Change its **Class** to **DetailViewController** (via the **Identity inspector** tab).
- For ease of reference, change the new scene's name from **Detail View Controller** to **Detail** by clicking on the yellow circle for the view controller on the Document Outline and clicking again to be able to edit the name.



Editing the scene name to give it a simpler name

- Similarly, rename the previously added **Search View Controller** scene to **Search**.
- Set the **Background** color of the Detail scene's 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.

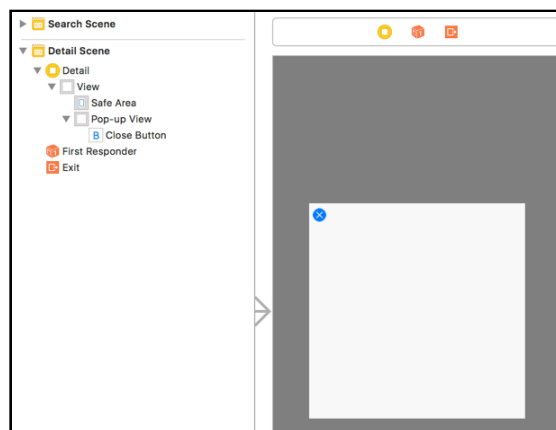
- In the **Attributes inspector**, change the **Background** color of this new view to **white, 95% opaque**. This makes it appear slightly translucent, just like navigation bars.
- With this new view still selected, go to the **Identity inspector**. For **Document - Label**, (the field with the hint text of “Xcode Specific Label”), type **Pop-up View**. You can use this field to give your views names, so they are easier to distinguish in the Document Outline in Interface Builder. (Now, instead of having multiple items called “View”, this particular view will display as “Pop-up View”.)



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 = 4 and Y = 2).
- 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 the title displayed in the Interface Builder; the user will never see that text.

The design should look something like this:



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

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

Show and hide the scene

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

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

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

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

► Control-drag from the yellow circle at the top of the Search scene to the Detail scene 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:

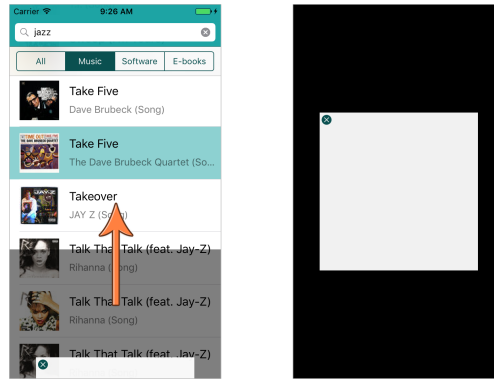
```
func tableView(_ tableView: UITableView,
               didSelectRowAt indexPath: IndexPath) {
    tableView.deselectRow(at: indexPath, animated: true)
    // Add the following line
    performSegue(withIdentifier: "ShowDetail", sender: indexPath)
}
```

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, do a search, and tap on a search result. Hmm, that doesn't look too good.

Even though you set the 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 it's still more work than necessary. (You'll see an example of view controller containment in an upcoming chapter 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.

Custom presentation controller

- 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.

Later 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 to the end of the file:

```
extension DetailViewController:
    UINavigationControllerTransitioningDelegate {

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

```
        presentedViewController: presented,  
        presenting: presenting)  
    }  
}
```

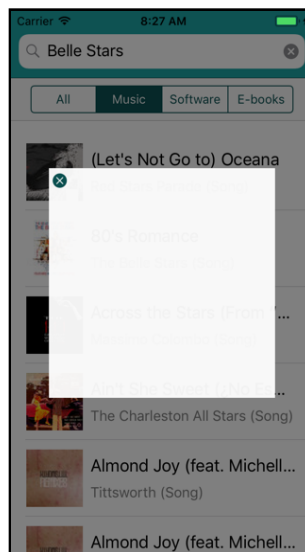
The methods from this delegate protocol tell UIKit what objects it should use to perform the transition to the Detail View Controller. It will now use your new `DimmingPresentationController` class instead of the standard presentation controller.

► Also add the following `init` method to `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 much 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 your 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.

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

Center the pop-up on all screens

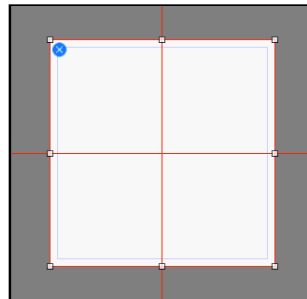
Now run the app on a simulator with a larger screen, like the iPhone 8 Plus. What happens? The Detail pop-up isn't properly centered in the screen anymore.

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 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**.

➤ Click **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

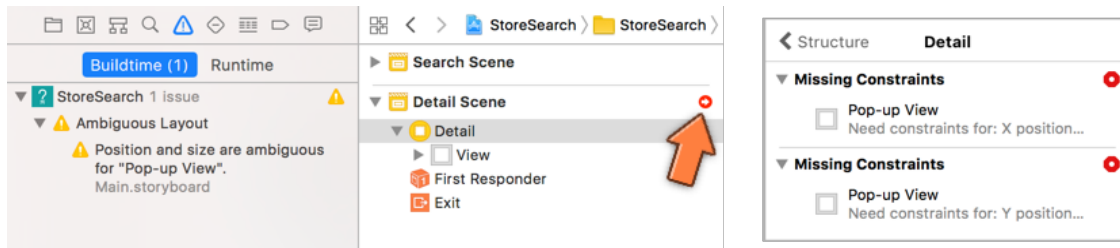
One small hiccup: these lines are supposed to be blue, not red. Whenever you see red or orange lines, Auto Layout has a problem.

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. 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 Document Outline to get a more detailed explanation of the errors. It's obvious that something's missing. You know it's not the position – the two alignment constraints are enough to determine that – so it must be the size.

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

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

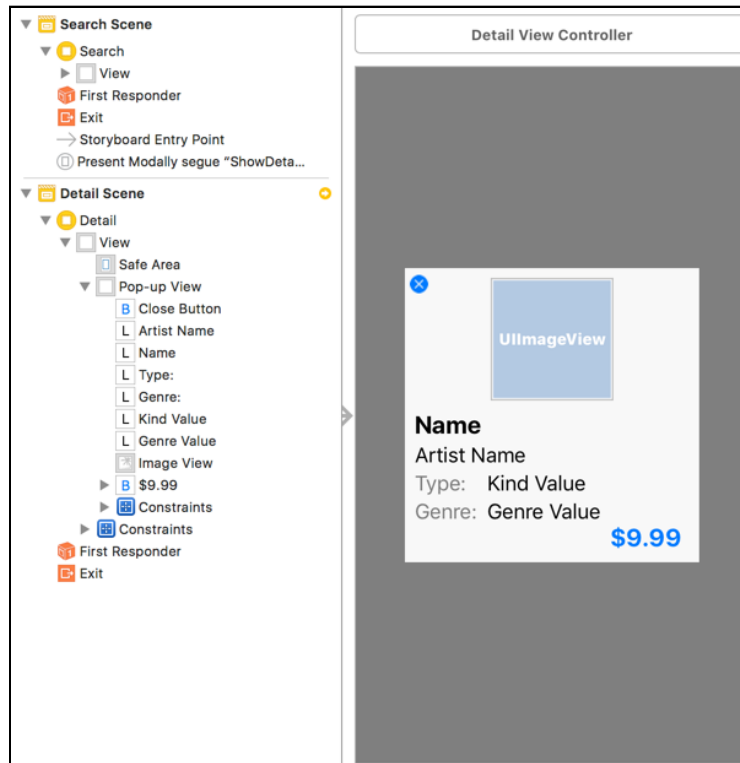
Now the lines turn blue and Auto Layout is happy.

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

Add 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

Add the controls

► Drag a new **Image View**, six **Labels**, and a **Button** on to the pop-up view and build a layout like the one from the picture.

Some suggestions for the dimensions:

Control	X	Y	Width	Height
Image View	70	8	100	100
Name label	8	116	220	24
Artist Name label	8	142	220	21
Type: label	8	165	43	21
Kind Value label	67	165	160	21
Genre: label	8	188	51	21
Genre Value label	67	188	160	21
\$9.99 button	164	208	68	24

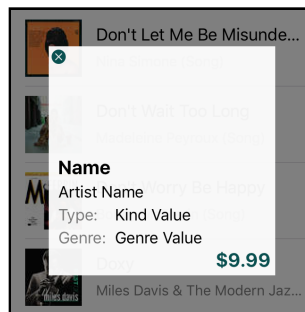
► The **Name** label's font is **System Bold 20**. Set **Autoshrink** to **Minimum Font Scale** so the font can become smaller if necessary to fit as much text as possible.

- The font for the **\$9.99** button is also **System Bold 20**. You will add a background image for this button in a bit.
- 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. Control-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.

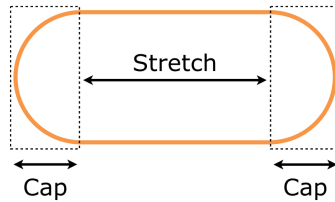


The new controls in the Detail pop-up

Stretchable images

The reason you did not put a background image on the price button yet is because 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.

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.



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

For *Bull's Eye* 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.

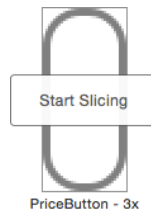
► Open **Assets.xcassets** and select the **PriceButton** image set.



The PriceButton image

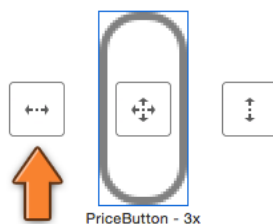
If you take a look at the image info, you will see that it is only 11 points wide. That means it has a 5-point cap on the left, a 5-point cap on the right, and a 1-point body that will be stretched out.

Click the **Show Slicing** button at the bottom of the central panel.



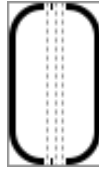
The Start Slicing button

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

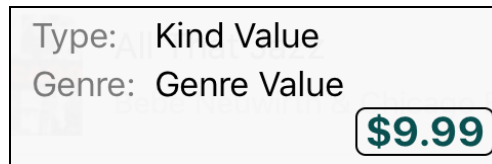
Each image is cut into three parts: the caps on the end and a one-pixel area in the middle that is the stretchable part. Now when you put this image onto a button or inside a UIImageView, it will automatically stretch itself to whatever size it needs to be.

Important: Do the above for *both* the 2x image and the 3x image.

► 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 depending on the price of the item. 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 tint color

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**.

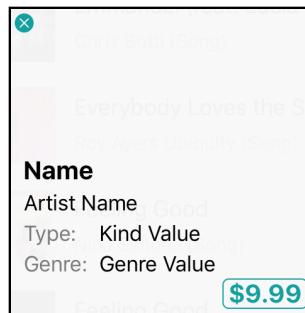
When you set the “template” rendering mode on an image, UIKit removes the original colors from the image and paints the whole thing in the tint color.

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 following line to `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.

Rounded corner views

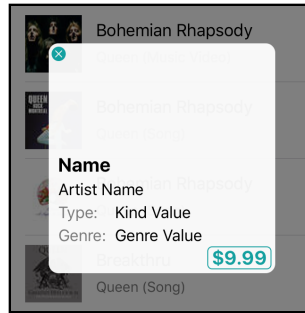
UIViews do their drawing using what’s known as a `CALayer` object. The CA prefix stands for Core Animation, which is the awesome framework that makes animations so easy on the iPhone. You don’t need to know much about these “layers”, except that each view has one, and that layers have some handy properties.

► 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

Tap gesture recognizer

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 - it will return `false` if the touch 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 a 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!

Show data in the 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 very different from what you've done in the previous apps!

There is more than one way to pull this off, but I like to do it by passing the `SearchResult` object to the `DetailViewController`.

Display selected item information in pop-up

► Add a property to **`DetailViewController.swift`** to store the passed in object reference:

```
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()`:

```
// MARK:- Helper Methods
func updateUI() {
    nameLabel.text = searchResult.name
```

```

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

    kindLabel.text = searchResult.type
    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()`.

► Add a call to the new method to the end of `viewDidLoad()`:

```

override func 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**:

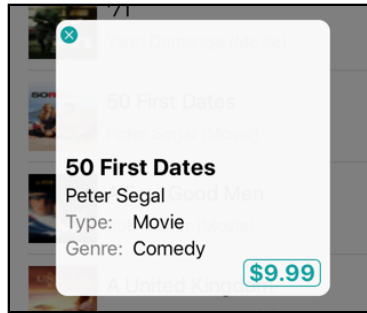
```

// MARK:- Navigation
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, now you're getting somewhere!



The pop-up with filled-in data

Show the price

You still need to show the price for the item and the correct currency.

► Add the following code to the end of `updateUI()`:

```
// Show price
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` previously to turn a `Date` object into human-readable text. Here you use `NumberFormatter` to do the same thing for numbers.

Previously, 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.

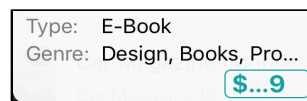
There are different rules for displaying various currencies, especially if you take the user's language and country settings into consideration. You could program all of these rules yourself, which is a lot of effort, or choose to ignore them. Fortunately, you don't have to make that tradeoff because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

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 format 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.

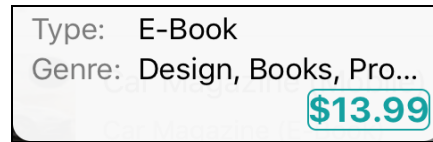
But buttons, like labels, are perfectly able to determine what their ideal size is based on the amount of text they contain. That’s called their *intrinsic content size*.

► Open the storyboard and with the price button selected, click the **Add New Constraints** button. Add two spacing constraints, one on the right and one on the bottom, both 8 points in size. Also add a **24** point Height constraint.

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 8 points. When the button needs to grow to accommodate larger prices, it will extend towards the left.
- Pinned to the bottom of the pop-up, also with a distance of 8 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.

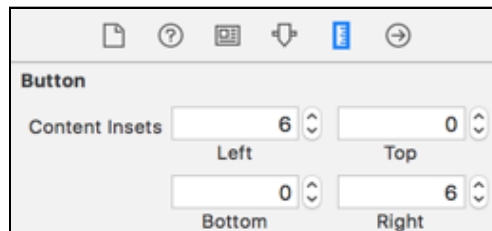
► Run the app again and pick an expensive product (something with a price over \$9.99; e-books are a good category for this).



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.

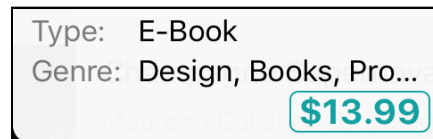
► Go to the **Size inspector** and find where it says **Content Insets**. Change **Left** and **Right** to 6.



Changing the content edge insets of the button

This adds 6 points of padding on the left and right sides of the button.

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



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

Note: After you added spacing constraints for the price button, you might have noticed that you started getting a compiler warning saying "Leading constraint is missing, which may cause overlapping with other views."

If you think about it, this makes sense since there is no leading constraint for the price button and if you were to add a new button to the left of the price button, you do run the risk of the price button accidentally expanding enough to overlap that hypothetical button. In this particular instance, it is not necessary to do anything since there won't be any other buttons for the price button to overlap. But if you wanted to remove the compiler warning, all you need to do is to add a leading constraint for the price button.

Navigate to the product page on iTunes

Tapping the price 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)  
    }  
}
```

- 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 returns 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 a device instead.)

Note: You haven't used UIApplication before, but every app has a UIApplication object 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.

Load artwork

For the Detail pop-up, you need to display a slightly larger, more detailed image than the one from the table view cell. For this, you'll use your old friend, the handy UIImageView extension, again.

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

```
var downloadTask: URLSessionDownloadTask?
```

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

```
// Get image  
if let largeURL = URL(string: searchResult.imageLarge) {  
    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.

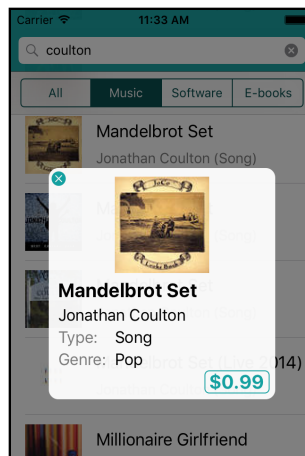
It's a good idea to cancel the image download if the user closes the pop-up before the image has been downloaded completely.

► To do that, 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 not done by then, you cancel it.

► 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.

You can find the project files for this chapter under **37 – The Detail Pop-up** in the Source Code folder.

Chapter 38: Polish the Pop-up

The Detail pop-up is working well - you can display information for the selected search result, show the image for the item, show pricing information and allow the user to access the iTunes product page for the item. You are done with the Detail pop-up can move on to the next item, right?

Well, not quite ... There are still a few things you can do to make the Detail pop-up more polished and user friendly.

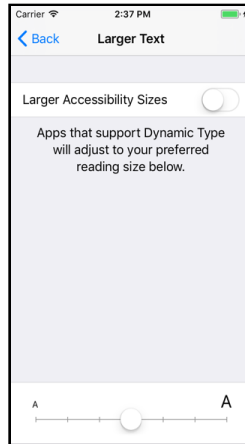
This chapter will cover the following:

- **Dynamic type:** Add support for dynamic type so that your text can display at a size specified by the user.
- **Gradients in the background:** Add a gradient background to make the Detail pop-up background look more polished.
- **Animation!:** Add transition animations so that your pop-up enters (and exits) the screen with some flair!

Dynamic Type

The iOS Settings app has an accessibility option (under **General** → **Accessibility** → **Larger Text**) 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 this setting both in your device and in the Simulator:



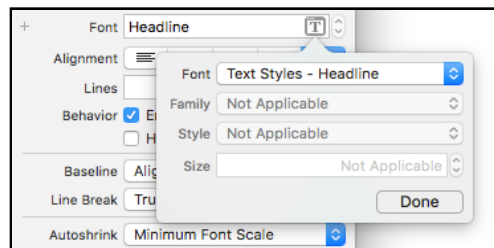
The Larger Text accessibility settings

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

Configure for Dynamic Type

To provide a better user experience for all users, whether their eyesight is good or bad, you'll change the Detail pop-up to use Dynamic Type for its labels.

► Open the storyboard and go to the **Detail** scene. Change the **Font** setting for the **Name** label to the **Headline** text style:



Changing the font to the dynamic Headline style

You can't pick a font size when selecting text styles - that depends on the user and the Larger Text setting they use on their device.

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

Auto Layout for Dynamic Type

Of course, if you don't know beforehand how large the label's font will be, you also won't know how large the label itself will end up being, especially if it sometimes may

have more than one line of text. You won't be surprised to hear that Auto Layout and Dynamic Type go hand-in-hand.

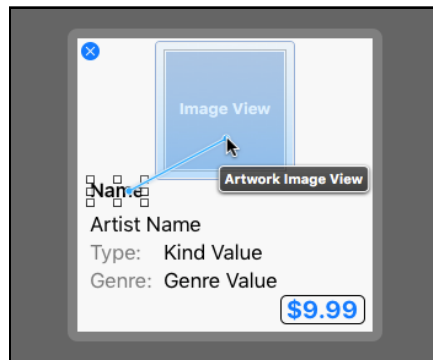
You want to make the name label resizable so that it can hold any amount of text at any possible font size, but it cannot go outside the bounds of the pop-up, nor overlap the labels below.

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

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

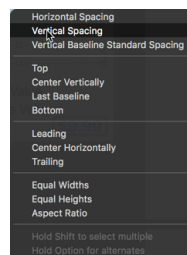
If you use the Add New Constraints button on the Name label, Interface Builder may decide to pin it to the bottom of the close button, which is weird. It makes more sense to pin the Name label to the image view instead. That's why you're going to use a different way to make constraints.

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



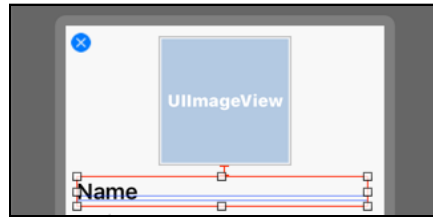
Control-drag to make a new constraint between two views

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:

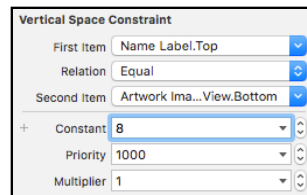


The new vertical space constraint

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.

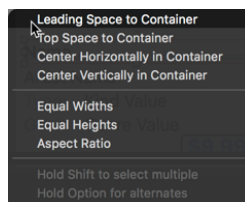
► Select the constraint (by carefully clicking it with the mouse or by selecting it from the Document Outline), then go to the **Size inspector** (or the Attributes inspector, they both show the same settings for layout constraints) and make sure that **Constant** is set to **8**.



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.

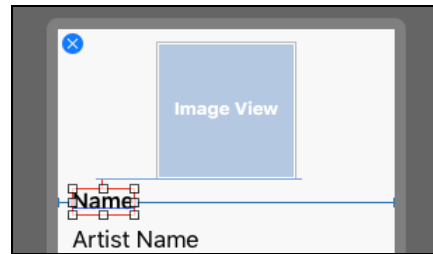
► Select the **Name** label again and **Control-drag** to the left and connect it to **Pop-up View**. Select **Leading Space to Container**:



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 set depend on the direction that you're dragging.

► Repeat the step but this time Control-drag to the right. Now choose **Trailing Space to Container**.



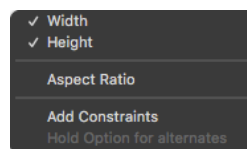
The constraints for the Name label

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**, **Control-drag** up to the Pop-up View, and choose **Top Space to Container**. That takes care of the Y-position.
- Repeat but now Control-drag to the left (or right) and 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.)
- Control-drag diagonally this time, but 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** (or, click on **Add Constraints** in the menu). (If you don’t see both options, make sure you Control-drag diagonally instead of straight up or sideways.)

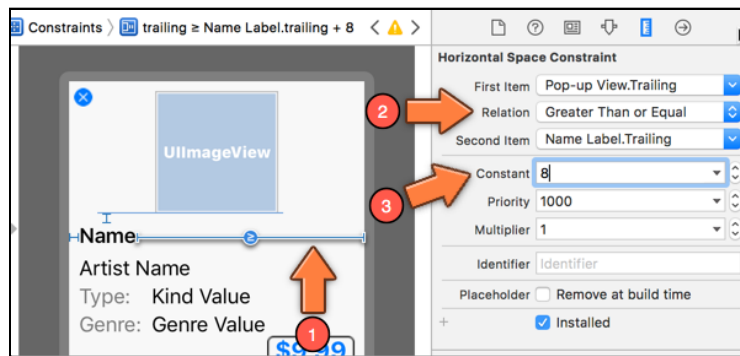


Adding multiple constraints at once

Now the image view and the Name label will have all blue bars.

There's one more thing you need to fix. Look again at that blue bar to 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 **8**.



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 8 points. This ensures there is at least a 8 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 for those languages too.

- Run the app and try it out:

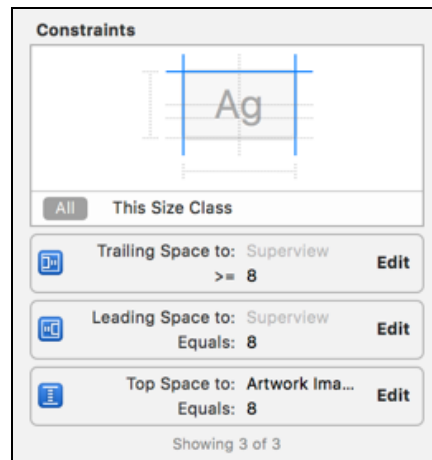


The text overlaps the other labels

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 Document Outline, 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

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

OK, let's make those changes...

- Select the **Artist Name** label and set its **Font** to the **Subhead** text style.
- Set the **Font** of the other four labels to the **Caption 1** text style. (You can do this in a single go if you multiple-select these labels by holding down the ⌘ key.)

Auto Layout for Artist Name

Let's pin the **Artist Name** label. Again you do this by Control-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 **8**.
- Pin it to the Name label with a **Vertical Spacing**. Change this to size **4**.

Auto Layout for Type

For the **Type:** label:

- Pin it to the left with a **Leading Space to Container**.
- 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 8**.
- Control-drag from **Kind Value** to **Type** and choose **Last 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.

Auto Layout for Genre

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**.
- On the right, pin it to the **Genre Value** label with a **Horizontal Spacing**. This should be a **8** point distance.

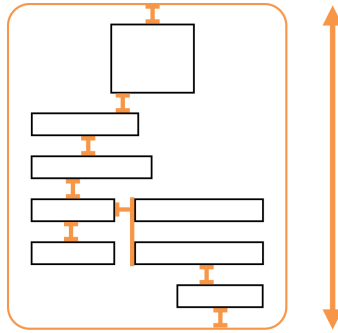
And finally, the **Genre Value** label:

- Pin it to the right with a **Trailing Space to Container, Greater Than or Equal 8**.
- Make a **Last 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 by selecting **Editor → Resolve Auto Layout Issues → Update Frames** from the Xcode menu. 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 Control-drag to make them is quite fast. With some experience you'll be able to whip together complex Auto Layout constraints in no time.

Auto Layout for Price button

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

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

While you might not notice this immediately, this introduces some Auto Layout constraint issues at this point - try clicking on the Genre: or Name labels and you'll see some constraints turn red.

This is because the Pop-up View still has a Height constraint that forces it to be 240 points high. But the labels (and image) and the vertical space constraints on these views don't add up to 240.

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

► If necessary (if you have any views with orange rectangles around them), 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.



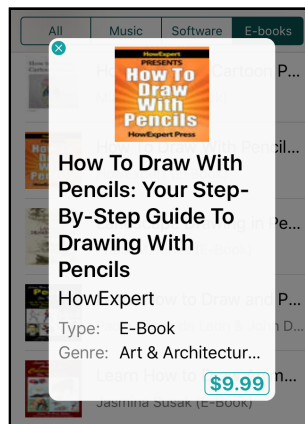
The text properly wraps without overlapping

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

Test Dynamic Type

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

Stack Views

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

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

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

Gradients in the background

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

It works well, but 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? (Also, an image would increase the size of your app and might also create some issues when you need to support larger screen sizes.) To pull this off, you will create your own `UIView` subclass.

The GradientView class

► 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 mostly opaque in the corners to mostly transparent in the center. Placed on a white background, it looks something like this:



What the GradientView looks like by itself

► Replace the contents of **GradientView.swift** with:

```
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. 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 what's known as a *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.

Generally speaking, it isn't optimal to create new objects inside your `draw()` method, such as gradients, especially if `draw()` is called often. In such cases 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. However, `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.

Use GradientView

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 to 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”.

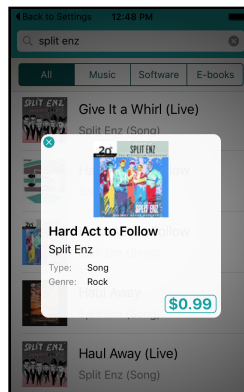
The container view is a new view that is placed on top of the `SearchViewController`, and it contains the views from the `DetailViewController`. So this piece of logic places the `GradientView` in between those two screens.

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 `viewDidLoad()` in **`DetailViewController.swift`**:

```
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 **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!

The animation controller class

- Add a new **Swift File** to the project, named **BounceAnimationController**.
- Replace the contents of the new file with:

```
import UIKit

class BounceAnimationController: NSObject,
    UINavigationControllerAnimatedTransitioning {

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

    func animateTransition(using transitionContext:
        UINavigationControllerContextTransitioning) {

        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.animateKeyframe(withRelativeStartTime: 0.334,
                           relativeDuration: 0.333, animations: {
        toView.transform = CGAffineTransform(scaleX: 0.9,
                                                y: 0.9)
    })
    UIView.animateKeyframe(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(using:)` – 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(using:)` – This performs the actual animation.

To find out what to animate, you look at the `transitionContext` parameter. This gives you a reference to a 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 animations: you set the initial state before the animation block, and `UIKit` will automatically animate any properties that get changed inside the closure. The difference from 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 it 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% of 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!

Use the new animation controller

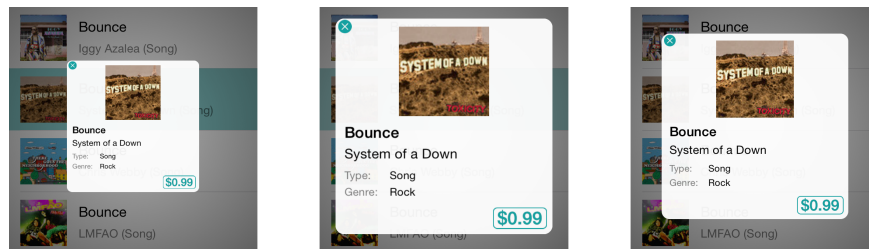
To use this animation in your app, you have to tell the app to use the new animation controller when presenting the Detail pop-up. That happens in the transitioning delegate inside **DetailViewController.swift**.

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

```
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.

Animate the background

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 end of `presentationTransitionWillBegin()`:

```
// Animate background gradient view
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 all 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 reverse: 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. :-)

Animate the pop-up exit

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 new file's 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 time = transitionDuration(using: transitionContext)
            UIView.animate(withDuration: time, 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()
}
```

This simply overrides the animation controller to be used when a view controller is dismissed.

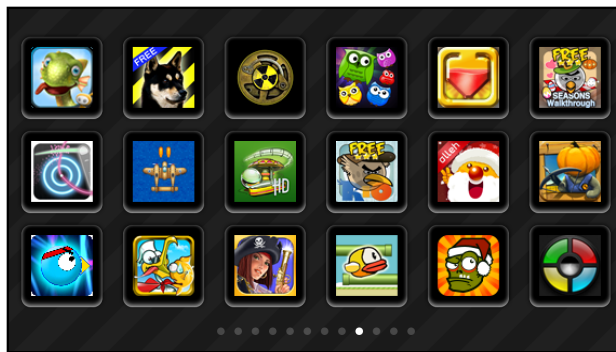
- Run the app and try it out. That looks pretty sweet if you ask me!
- If you're happy with the way the animations look, then commit your changes.

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.

You can find the project files for this chapter under **38 – Polish the Pop-up** in the Source Code folder.

Chapter 39: Landscape

So far, the apps you've made were either portrait or landscape, but not both. Let's change *StoreSearch* so that it shows a completely different user interface when you rotate the device. 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.

You'll cover the following in this chapter:

- **The landscape view controller:** Create a basic landscape view controller to make sure that the functionality works.
- **Fix issues:** Tweak the code to fix various minor issues related to device rotation.
- **Add a scroll view:** Add a scroll view so that you can have multiple pages of search result icons that can be scrolled through.

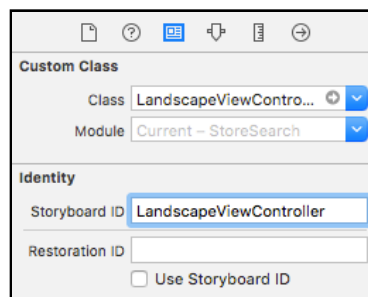
- **Add result buttons:** Add buttons in a grid for the search results, to the scroll view, so that the result list can be scrolled through.
- **Paging:** Configure scrolling through results page-by-page rather than as a single scrolling list.
- **Download the artwork:** Download the images for each search result item and display it for each item in the scroll view.

The landscape view controller

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

The view controller in the storyboard

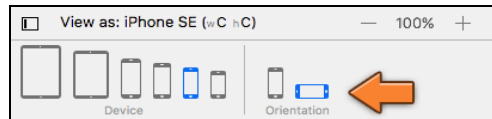
- Add a new file to the project using the **Cocoa Touch Class** template. Name it **LandscapeViewController** and make it a subclass of **UIViewController**.
- In Interface Builder, drag a new **View Controller** on to the canvas.
- In the Document Outline, click on the yellow circle for the view controller and change its name to **Landscape**.
- 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. Instead, 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 uniquely identify this particular view controller in the storyboard.

- Use the **View as:** panel to change the orientation to landscape.

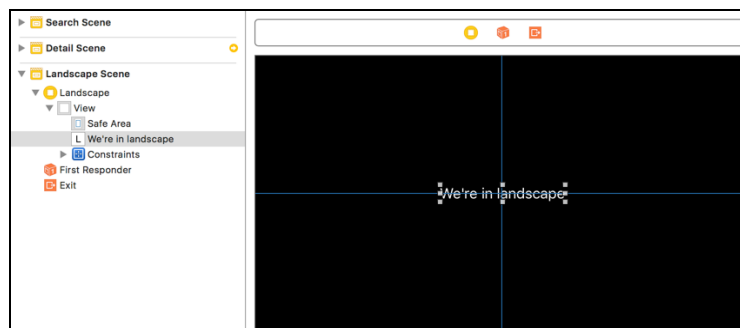


Changing Interface Builder to landscape

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

- Change **View - Background** 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.
- Change the label's **Label - Color** to **White**, and if not all the text is showing use the **Editor** → **Size to Fit Content** menu option (or the ⌘= shortcut) to resize the label to fit its content.
- Use the **Align** Auto Layout menu to center the label horizontally and vertically.

Your design should look something like this:



Initial design for the Landscape scene

Show the landscape view on device rotation

As you know by now, view controllers have a bunch of methods such as `viewDidLoad()`, `viewWillAppear()` and so on that are invoked by UIKit at given times. There is also a method that is invoked when the device is rotated. 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:

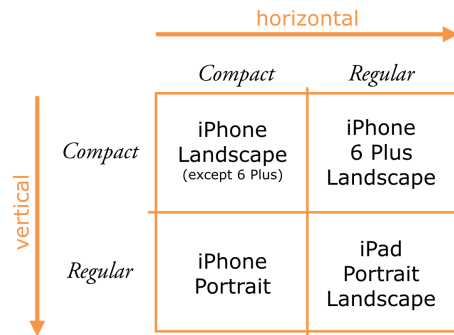
- The horizontal size class
- The vertical size class
- The display scale (is this a Retina screen or not?)
- The user interface idiom (is this an iPhone or iPad?)
- The preferred Dynamic Type font size
- And a few other things

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 “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 Plus models, that is.

In landscape, the horizontal size class on the Plus is *regular*. That's because the larger dimensions of the iPhone Plus devices 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, when devices with larger screens became available, that became inconvenient – you often want one area of the screen to be controlled by one view controller and a second area by a separate 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 eminently 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 landscapeVC: LandscapeViewController?
```

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

➤ Add the following method:

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

In previous apps you called `present(animated:completion:)` or made a segue to show a 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 statement codifies this requirement. If it should happen that `landscapeVC` is not `nil`, then you're already showing the landscape view and you 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 instantiate the view controller manually. This is why you filled in that Storyboard ID field in the Identity inspector.

The `landscapeVC` 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. Add the landscape controller's view as a subview. This places it on top of the table view, search bar and segmented control.
 - b. 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 the parent - it isn't 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.

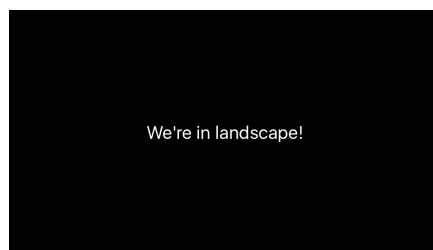
One of the reasons you're not using a modal segue for the Landscape screen in this app, even though it is a full-screen view controller, is that the Detail pop-up already is modally presented and this could potentially cause conflicts. Besides, I wanted to show you a fun alternative to modal segues.

- 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 rotate your iPhone or the Simulator to landscape.



The Simulator after flipping to landscape

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

This is not doing any animation just yet. As always, first get it to work right, and *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** (on the Simulator only) to hide the keyboard manually.

Switch back to the portrait view

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

► Replace the *method stub*, which is basically a method name with no implementation code, that you added earlier with the following implementation to hide the landscape view controller:

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

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

First, you call `willMove(toParentViewController:)` 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. Switching back to portrait should remove the black landscape view.

Note: If you press **⌘-right** (or **⌘-left**) 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 the app keeps thinking it's in landscape.

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

Animate the transition to landscape

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 in **SearchViewController.swift** as follows:

```
func showLandscape(with coordinator:
                    UINavigationControllerTransitionCoordinator) {
    . . .
    if let controller = landscapeVC {
        controller.view.frame = view.bounds
        controller.view.alpha = 0           // New line

        view.addSubview(controller.view)
        addChildViewController(controller)
        // Replace all code after this with the following lines
        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 transparent (`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 you don't use it here and so, you use the `_` wildcard to ignore it.

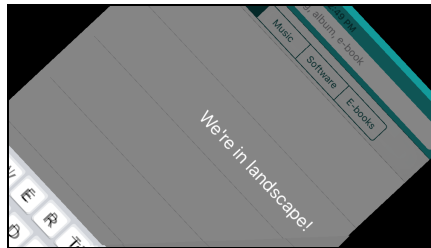
Animate the transition from landscape

► Make similar changes to `hideLandscape(with:)`:

```
func hideLandscape(with coordinator:
    UINavigationControllerTransitionCoordinator) {
    if let controller = landscapeViewController {
        controller.willMove(toParentViewController: nil)
        // Replace all code after this with the following lines
        coordinator.animate(alongsideTransition: { _ in
            controller.view.alpha = 0
        }, completion: { _ in
            controller.view.removeFromSuperview()
            controller.removeFromParentViewController()
            self.landscapeVC = nil
        })
    }
}
```

This time you fade out the view. 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.

Note: The order of operations for removing a child view controller is exactly the reverse of 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.

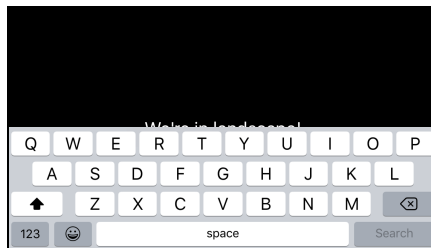
You can find these rules in the API documentation for `UIViewController`.

Fix issues

There are two more small tweaks that you need to make.

Hide the keyboard

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 this 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 rotate 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.

Hide the Detail pop-up

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.

The Detail pop-up is presented modally via 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 though: 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.

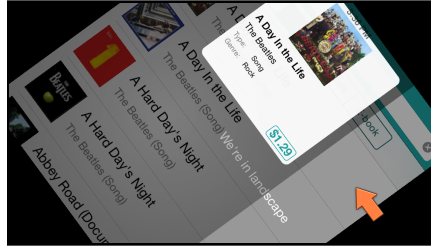
► Add the following code to the end of the `animate(alongsideTransition:)` closure in `showLandscape(with:)`:

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

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

Fix the gradient view

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. But do this only after the Detail pop-up has appeared. Some (all?) of the animation completions don't appear to fire with slow animations on and so you might not get the expected behavior otherwise.)

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 the window. But you didn't create GradientView in Interface Builder... 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 *MyLocations* 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 for some other reason).

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 is rotated.

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

Tweak the animation

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 for 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 dismissStyle = 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 `dismissStyle` 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 it keeps using the animation you're already familiar with:

```
@IBAction func close() {  
    dismissStyle = .slide // Add this line  
    dismiss(animated: true, completion: nil)  
}
```

- Add a new **Swift File** to the project, named **FadeOutAnimationController**. This will handle the animation for the `.fade` style.
- Replace the source code of this 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 time = transitionDuration(using: transitionContext)  
            UIView.animate(withDuration: time, animations: {  
                fromView.alpha = 0  
            }, completion: { finished in  
                transitionContext.completeTransition(finished)  
            })  
        }  
    }  
}
```

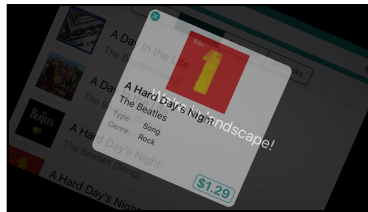
This is mostly the same as the other animation controllers. The actual animation simply sets the view's `alpha` value to 0 in order to fade it out.

► Switch to **DetailViewController.swift** and in the extension for the transitioning delegate, change the method that returns the animation controller for dismissing the pop-up to the following:

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

Instead of always returning a new `SlideOutAnimationController` instance, it now looks at the value from `dismissStyle`. If it is `.fade`, then it returns an instance of the new `FadeOutAnimationController` object.

► 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.

Add a scroll view

If an app has more content to show than can fit on the screen, you can use a **scroll view**, which allows the user to, as the name implies, scroll through the content horizontally and/or vertically.

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

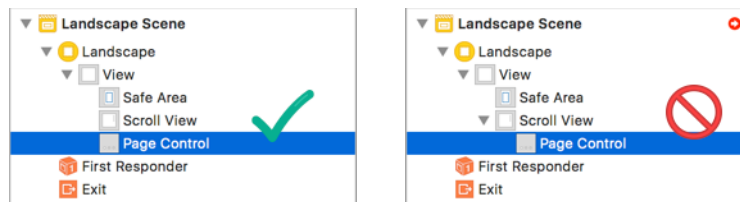
In this section, you’ll use a scroll view of your own, in combination with a **paging control**, to show the artwork for all the search results, even if there are more images than can fit on the screen at once.

Add the scrollview to the storyboard

- Open the storyboard and delete the label from the Landscape scene.
- Now, drag a **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.

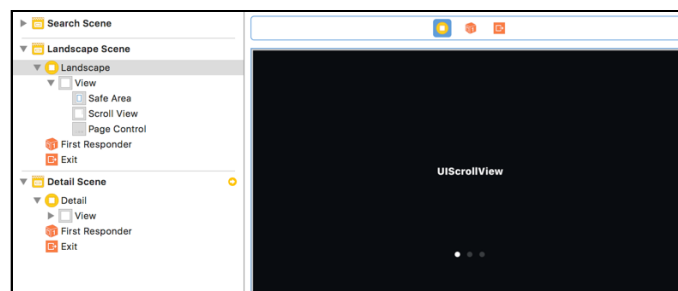
Important: Do not place the Page Control *inside* the Scroll View. They should be at the same level in the view hierarchy:



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 in the Document Outline.

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



The final design of the Landscape scene

Disable Auto Layout for a view controller

The other view controllers all employ Auto Layout to resize them to the dimensions of the user's screen, but here, you're going to take a different approach. Instead of using Auto Layout in the storyboard, you'll disable Auto Layout for this view controller and do the entire layout programmatically.

You do need to hook up the 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()` with:

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Remove constraints from main view
    view.removeConstraints(view.constraints)
    view.translatesAutoresizingMaskIntoConstraints = true
    // Remove constraints for page control
    pageControl.removeConstraints(pageControl.constraints)
    pageControl.translatesAutoresizingMaskIntoConstraints = true
    // Remove constraints for scroll view
    scrollView.removeConstraints(scrollView.constraints)
    scrollView.translatesAutoresizingMaskIntoConstraints = true
}
```

Remember how, if you don't add 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 all the visible views in the view controller first.

You also do `translatesAutoresizingMaskIntoConstraints = true`. This 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. You wouldn't like it when it's 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, possibly causing your app to crash.

Custom scroll view layout

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

► 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 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.

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.

Add a background to the scroll view

For the scroll view to do anything, you have to add some content to it. But for the moment, just to see that the scroll view is there, let's add a background to it.

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

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

This puts an image on the scroll view's background. 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 as a color.

If you take a peek at the **LandscapeBackground** image in the asset catalog, you'll see that it is a small square. When you set this image as a pattern image for the background, the image repeats to cover the entire area. Tile-able images can be used anywhere where you can use a `UIColor`.

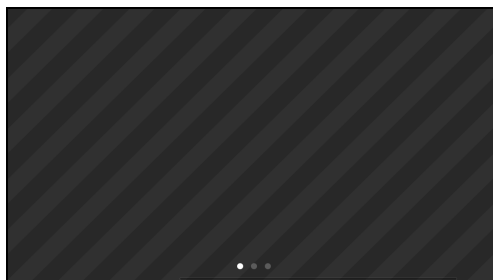
► Also, add the following line to `viewDidLoad()`:

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

It is very important to set the `contentSize` property when dealing with scroll views. This tells the scroll view how big the content area for the scroll view is - a scroll view's inside (the content area), can be bigger than its actual bounds. If the content area is bigger than the scroll view's bounds, that's when the scroll view allows you to scroll.

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:



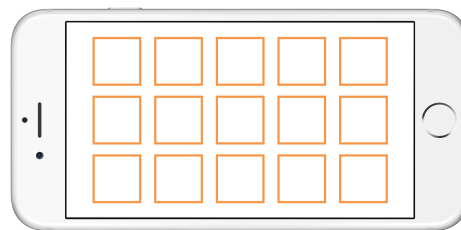
The scroll view now has a background image and it can scroll

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 Document Outline drag the Page Control below the Scroll View.

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.

Add result buttons

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. 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 partially.

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

That's a lot of different possibilities!

You need to add the logic to `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 display the artwork via that `UIButton`.

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

Pass the search results to the landscape view

► Let's add a property for this to **`LandscapeViewController.swift`**:

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

Initially, this will be an empty array. `SearchViewController` replaces it with 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
    }
    . . .
```

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

The view controller will read 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 no buttons will be created. The order in which you do things matters!

► Switch back to **`LandscapeViewController.swift`**. Remove the line that sets `scrollView.contentSize` from `viewDidLoad()`. That was just for testing.

Now let's go make those buttons.

Initial configuration

► Add a new instance variable:

```
private var firstTime = true
```

The purpose for this variable will become clear in a moment.

Private parts

You declared the `firstTime` instance variable as `private`. This is 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` when `LandscapeViewController` isn't expecting the change.

We haven't talked much about the distinction between *interface* and *implementation* yet, but what an object shows to the outside is different from what it has on the inside. That's done on purpose because its internals – the implementation details – should not be of interest to anyone else, and are often even dangerous to expose (messing around with internal settings 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. 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 end of `viewWillLayoutSubviews()` (but note that the `tileButtons(_:)` method has not been implemented yet):

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

This calls a new method, `tileButtons(_:)`, that performs the necessary 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 this, 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 the view should be. Only after `viewDidLoad()` is done does the view get resized to fit the actual screen.

So you can't use `viewDidLoad()` for this. 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.

Calculate the tile grid

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

```
// MARK:- Private Methods
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 viewWidth = scrollView.bounds.size.width

    switch viewWidth {
    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 should 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. 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. 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. 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. 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: Shouldn'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.)

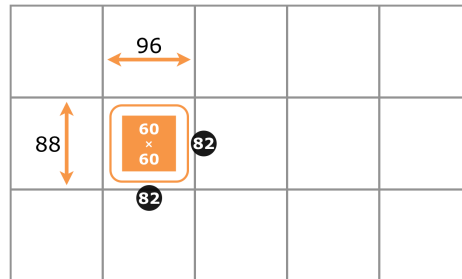
From now on, you'll keep adding more code to the end of `tileButtons()` (where the `TODO` comment is) till the method is complete.

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

```
// Button size
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

Add buttons

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

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

```
// Add the buttons
var row = 0
var column = 0
var x = marginX
for (index, result) 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 properties are all `CGFloat` 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 to the `UIScrollView` as a subview. After the first 18 or so buttons (depending on the screen size), this places any subsequent buttons 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 view these 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, you can have those statements on separate lines, if you so prefer.

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, result) in searchResults.enumerated() {
```

This `for in` loop steps through the `SearchResult` objects from the array, but with a twist. By calling the array's `enumerated()` method, 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, result)`. 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:

```
// Set scroll view content size
let buttonsPerPage = columnsPerPage * rowsPerPage
let numPages = 1 + (searchResults.count - 1) / buttonsPerPage
scrollView.contentSize = CGSize(
    width: CGFloat(numPages) * viewWidth,
    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 (you’ll enable this feature shortly), rather than simply scroll. So, you should always make the content width a multiple of the screen width (480, 568, 667 or 736 points). You can then determine how many pages you need with a simple formula.

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 `Ints`, which makes `numPages` an `Int` too.

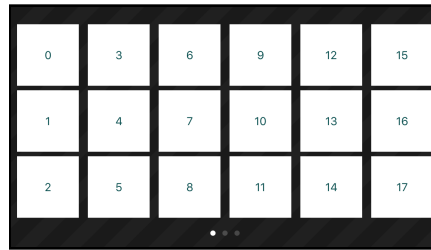
That’s why the formula is `1 + (searchResults.count - 1) / buttonsPerPage`.

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 to verify 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 number of pages.

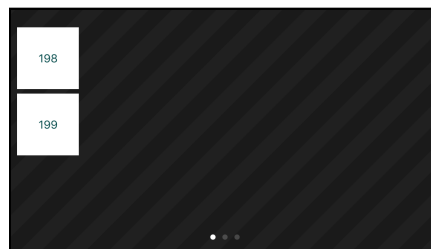
Note: Xcode currently gives a warning “Immutable value ‘result’ was never used; consider replacing with ‘_’ or removing it”. That warning will go away once you use the `result` variable in the next section.

► Run the app, do a search, and rotate 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 test these situations 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. You'll add a "Nothing Found" label to this screen too, in a bit.

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 moved 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 in its springboard. Many other apps use the effect too, for example, the Weather app uses paging to flip between the cards for different cities.

Enable scroll view paging

► Go to **Landscape** scene 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 of the screen.

Configure the page control

► Switch to **LandscapeViewController.swift** and 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 end 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.

Connect the scroll view and page control

- Add this new extension to the end of **LandscapeViewController.swift**:

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

This is a `UIScrollViewDelegate` method. 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". So, you have to calculate this yourself. If the content offset gets beyond halfway on the page ($\text{width}/2$), the scroll view will move 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 for it.

- Add the action method:

```
// MARK:- Actions  
@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, for the **Landscape** scene, **Control-drag** from the Scroll View to the view controller and select **delegate**.
- Also **Control-drag** from the Page Control to the view controller and select **pageChanged:** under Sent Events.
- Try it out, the page control and the scroll view should now be in sync.

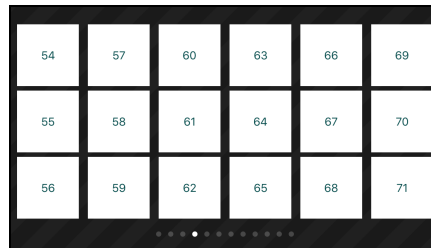
The transition from one page to another after tapping in the page control is still a little abrupt, though. An animation would help here.

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

You can simply wrap the code from the action method 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.

Download the artwork

First let's give the buttons a nicer look.

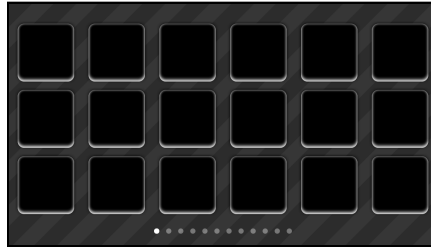
Set button background

► Replace the button creation code in `tileButtons()` (in **LandscapeViewController.swift**) with:

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

Instead of a regular button, you now make a `.custom` one, and you give 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

Display button images

Now you have to download the artwork images (if they haven't already been downloaded and cached 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.imageSmall) {
        let task = 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)
                    }
                }
            }
        }
        task.resume()
    }
}
```

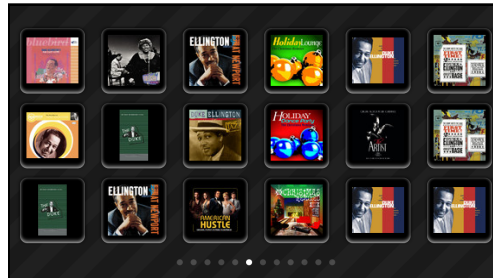
This looks very much like what you did in the `UIImageView` extension.

First you get a `URL` instance 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: result, 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 `result` 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 app but in the mean time, you can replace it by the `_` wildcard symbol to stop Xcode from complaining.

Clean up

It's always a good idea to clean up after yourself, in life as well as 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 not handled properly.

To avoid ownership cycles, you capture the button with a weak reference. When `LandscapeViewController` is deallocated, so are the buttons. So, 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 are not needed. Otherwise, it's just wasting bandwidth and battery life, and users don't take too kindly to apps that do this.

- Add a new property to **LandscapeViewController.swift**:

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

This array will keep track of all the active `URLSessionDownloadTask` objects.

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

```
downloads.append(task)
```

- And finally, add a `deinit` method to cancel any operations that are still on the way:

```
deinit {  
    print("deinit \(self)")  
    for task in downloads {  
        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 are 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 in size.

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

You can find the project files for this chapter under **39 – Landscape** in the Source Code folder.

Chapter 40: Refactoring

Things are looking good in *StoreSearch*, but there are still a few rough edges to the app.

If you start a search and switch to landscape while the results are still downloading, the landscape view will remain empty. You can reproduce this situation by artificially slowing down your network connection using the Network Link Conditioner tool.

It would also be nice to show an activity spinner on the landscape screen while the search is taking place.

You will polish off some of these rough edges in this chapter. You will cover the following:

- **Refactor the search:** Refactor the code to put the search logic into its own class so that you have centralized access to the search state and results.
- **Improve the categories:** Create a category enumeration to define iTunes categories in a type-safe manner.
- **Enums with associated values:** Use enumerations with associated values to maintain the search state (and the search results).
- **Spin me right round:** Add an activity indicator to the landscape view. Also add a network activity indicator to the app.
- **Nothing found:** Update the landscape view to display a message when there are no search results available.
- **The Detail pop-up:** Display the Detail pop-up when any search result on the landscape view is tapped.

Refactor the search

So how can `LandscapeViewController` tell what state the search is in? Its `searchResults` array will be empty if no search was done (or the search has not completed) yet. Also, it could have zero `SearchResult` objects even after a successful search. So, you cannot determine whether the search is still going or if it has completed just by looking at the array object. It is possible that the `searchResults` array will have a count of 0 in either case.

You need a way to determine whether a search is still going on. 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!

The Search class

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

This is a pretty comprehensive change to the code and there is always a risk that it won't work as you hoped. By making the changes in a new branch, you can commit your changes without messing up the master branch. (Plus, you can revert back to the master branch if the changes don't work out.) 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 it compiles without errors, you will move all the logic over. Baby steps!

Move code over

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 properties:

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

And replace them with this one:

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

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

► Move the following methods over to **`Search.swift`**:

- `iTunesURL(searchText:category:)`
- `parse(data:)`

► 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, you still reload the table view (to show the activity spinner) and hide 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  
    }  
}
```

Similar to the above, find the other places in code where the relevant properties have moved and make the necessary changes. (If you aren't sure of where to make the changes, look for Xcode errors - for this step, once you make all the changes correctly, the code will compile again without any errors.)

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

```
controller.searchResults = search.searchResults
```

To:

```
controller.search = search
```

This line will give an error after you make the change, but you'll fix that next.

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.

Add the search logic back in

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 (you can use that temporary file from earlier, 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
            // Was the search cancelled?
            if let error = error as NSError?, error.code == -999 {
                return
            }

            if let httpResponse = response as? HTTPURLResponse,
                httpResponse.statusCode == 200, let data = data {
                self.searchResults = self.parse(data: data)
                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 logic as before, except all the user interface code 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 Console shows a “Success!” message but the table view does not reload and the spinner keeps spinning for 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.

The `SearchComplete` closure

So, let's create your own closure!

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

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

The `typedef` declaration 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 declare 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 a `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 the same as the 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 httpResponse = response as? . . . {
                self.isLoading = false
                success = true                                         // instead of return
            }
            if !success {                                             // new
                self.hasSearched = false
                self.isLoading = false                                // new
            }
            // New code block - add the next three lines
            DispatchQueue.main.async {
                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           // Begin new code
        if !success {
            self.showNetworkError()
        }
        self.tableView.reloadData()
    })           // End new code

    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, right? 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 how it is supposed to work.

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

Improve 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`.

Represent the category as an enum

There are only four possible search categories, so this sounds like a 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 values. 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  
}
```

That enum does not associate numbers with 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 values 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 have associated numbers.

Use the Category enum

► 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 to pass it the value 4 or 99 or -1 anymore. 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 kind: String
    switch category {
    case .all: kind = ""
    case .music: kind = "musicTrack"
    case .software: kind = "software"
    case .ebooks: kind = "ebook"
    }

    let encodedText = . . .
```

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 the category parameter 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 a “kind” 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 type property to the `Category` enum:

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

```

    case ebooks = 3

    var type: 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. `type` 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 kind = category.type
    let encodedText = . . .
}

```

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

Convert an Int to 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: {
            . . .
        })
        . . .
    }
}

```

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.

Enums with associated values

Enums are pretty useful for restricting 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...

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).

Consolidate search state

You can improve upon things 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 an **associated value**, which is an array of `SearchResult` objects.

This array is only important when the search is successful. In all the other cases, there are no search results and the array is empty (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.

Use the new state enum

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 so. 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 (or setting) new values to this variable may only happen inside the Search class.

► Change `performSearch(for:category:completion:)` to use this new variable:

```
func performSearch(for text: String, category: Category,
                  completion: @escaping SearchComplete) {
    if !text.isEmpty {
        dataTask?.cancel()
        // Remove the next 3 lines and replace with the following
        state = .loading
        .
        .
        .
        dataTask = session.dataTask(with: url, completionHandler: {
            data, response, error in

            var newState = State.notSearchedYet           // add this
            .
            .
            .
            if let httpResponse = response . . . {
                // Replace all code within this if block with following
                var searchResults = self.parse(data: data)
                if searchResults.isEmpty {
                    newState = .noResults
                } else {
                    searchResults.sort(by: <)
                    newState = .results(searchResults)
                }
                success = true
            }
            // Remove "if !success" block
            DispatchQueue.main.async {
                self.state = newState                     // add this
                completion(success)
            }
        })
        dataTask?.resume()
    }
}
```

Instead of the old variables `isLoading`, `hasSearched`, and `searchResults`, this code now only changes `state`.

Note: You don't update `state` directly, but instead, use a new local variable `newState`. Then at the end, in the `DispatchQueue.main.async` block, you transfer the value of `newState` to `self.state`. The reason for doing this the long way round is that `state` must only be changed by the main thread, or it can lead to a nasty and unpredictable bug known as a *race condition*.

When you have multiple threads trying to use the same variable at the same time, the app may do unexpected things and crash. In our app, the main thread will try to use `search.state` to display the activity spinner in the table view – and that can

happen at the same time as `URLSession`'s completion handler, which runs in a background thread. We have to make sure these two threads don't get in each other's way!

Here's how the new logic works:

There is a lot that can go wrong between performing the network request and parsing the JSON. By setting `newState` to `.notSearchedYet` (which doubles as the error state) and `success` to `false` at the start of the completion handler, you assume the worst – always a good idea when doing network programming – unless there is evidence otherwise.

That evidence comes when the app is able to successfully parse the JSON and create an array of `SearchResult` objects. If the array is empty, `newState` becomes `.noResults`.

The interesting part is when the array is *not* empty. After sorting it like before, you do `newState = .results(searchResults)`. This gives `newState` the value `.results` and also associates the array of `SearchResult` objects with it. You no longer need a separate instance variable to keep track of the array; the array object is intrinsically attached to the value of `newState`.

Finally, you copy the value of `newState` into `self.state`. As I mentioned, this needs to happen on the main thread to prevent race conditions.

Update other classes to use the state enum

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 properties.

► In **SearchViewController.swift**, replace `tableView(_:numberOfRowsInSection:)` with:

```
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 a bit 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.

► Replace `tableView(_:cellForRowAt:)` with:

```
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 happens 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`. Since it would be a bug if the code ever got there, you can use the built-in `fatalError()` function to help catch such a situation.

➤ 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 for all the other cases, this method returns `nil`. And for the `.results` case, you don't need to bind the results array because you're not using it for anything here.

➤ And finally, change `prepare(for:sender:)` 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` block 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()`. The reason you don't use a `if case` condition here is because you'll be adding additional code to the other cases soon. But, 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 rotate to landscape while the search is still taking place, the app really ought to show an animated spinner to let the user know that an action is taking place. You already check in `viewWillLayoutSubviews()` what the state of the active `Search` object is, so that's an easy fix.

Show an activity indicator in landscape mode

► In `LandscapeViewController.swift`, add a new method to display an activity indicator:

```
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()
}
```

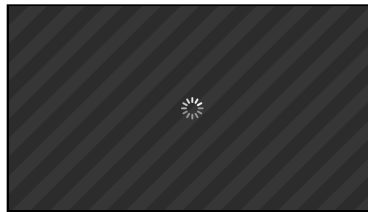
This creates a new `UIActivityIndicatorView` object (a big white one this time), puts it in the center of the screen, and starts animating it.

You give the spinner the tag 1000, so you can easily remove it from the screen once the search is done.

► In `viewWillLayoutSubviews()` change the `.loading` case in the switch statement to call this new method:

```
case .loading:
    showSpinner()
```

► Run the app. After starting a search, quickly rotate the phone to landscape. You should now see a spinner:



A spinner indicates a search is still taking place

Note: In the new method you add `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 will be 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.

Hide the landscape spinner when results are found

This is all great, but the spinner doesn't disappear when the actual search results are received. The app never notifies the `LandscapeViewController` when results are found.

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:

```
// MARK:- Public 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 and used that, 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.landscapeVC?.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.landscapeVC` 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.landscapeVC` 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.landscapeVC`, 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: if you don't want to use the Network Link Conditioner, 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.

Show the network activity indicator

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**, add this import:

```
import UIKit
```

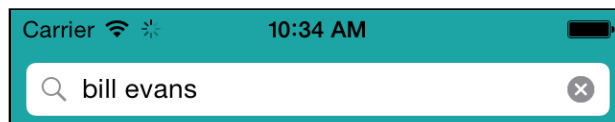
► Add the following line to `performSearch(for:category:completion:)`, just before starting the search:

```
func performSearch(for text: String, category: Category,
                  completion: @escaping SearchComplete) {
    if !text.isEmpty {
        dataTask?.cancel()
        UIApplication.shared.isNetworkActivityIndicatorVisible =
        true
        . . .
    }
}
```

This makes the network activity indicator visible in the app's status bar. To turn it off again, add the following line to `DispatchQueue.main.async` at the end of `performSearch(for:category:completion:)`:

```
UIApplication.shared.isNetworkActivityIndicatorVisible = false
```

► 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.

► First, add the following method to **LandscapeViewController.swift**:

```
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)
}
```

You first create a `UILabel` object and give it text and a color. The `backgroundColor` property is set to `UIColor.clear` to make the label transparent.

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
```

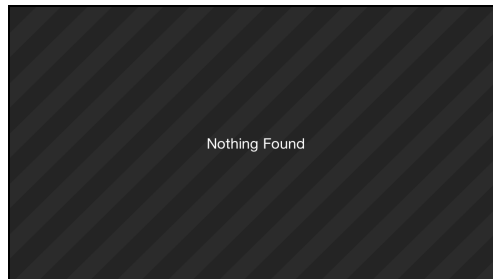
If you divide a number such as 11 by 2 you get 5.5. The `ceil()` function rounds up 5.5 to make 6, and then you multiply by 2 to get a final value of 12. This formula always gives you the next even number if the original is odd. (You only need to do this because these values have type `CGFloat`. If they 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 screen sizes.

► Inside the switch statement in `viewWillLayoutSubviews()`, call the new method from the case for `.noResults`:

```
case .noResults:
    showNothingFoundLabel()
```

► 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 if 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:

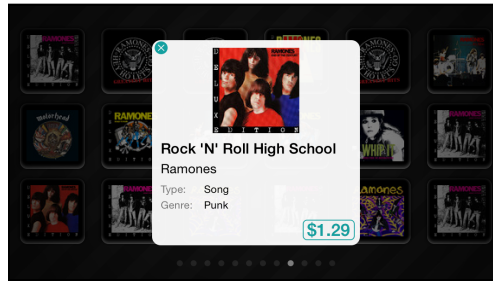
```
switch search.state {
case .notSearchedYet, .loading:
    break
case .noResults:
    showNothingFoundLabel()
case .results(let list):
    tileButtons(list)
}
```

Now you should have all your bases covered.

The Detail pop-up

The landscape view is that much more functional after all the refactoring and changes. But there's still one more thing left to do. The landscape search results are not buttons for nothing.

The app should show the Detail pop-up when you tap an item, 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.

► First, still in **LandscapeViewController.swift** add the method to be called when a button is tapped:

```
@objc 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 make the connection via code, so you can skip the @IBAction annotation.

Also note that the method has the @objc attribute - as you learnt previously with *MyLocations*, you need to tag any method that is identified via a #selector with the @objc attribute. (So, that would seem to indicate that you'll be calling this new method using a #selector, right?)

Pressing the button simply triggers a segue, and you'll get to the segue part in a moment. But first, you should hook up the buttons to the above method.

► 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 the `buttonPressed()` method when it gets tapped.

► Next, add the `prepare(for:sender:)` method to handle the segue:

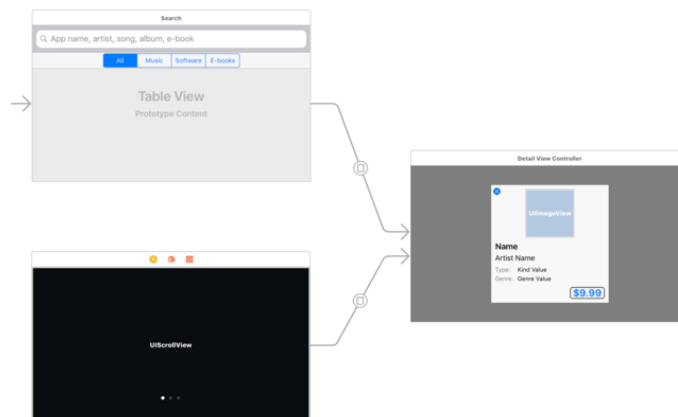
```
// MARK:- Navigation
override func prepare(for segue: UIStoryboardSegue,
                      sender: Any?) {
    if segue.identifier == "ShowDetail" {
        if case .results(let list) = search.state {
            let detailViewController = segue.destination
                                   as! DetailViewController
            let searchResult = list[(sender as! UIButton).tag - 2000]
            detailViewController.searchResult = searchResult
        }
    }
}
```

This is almost 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 have a segue in the storyboard.

► Go to the Landscape scene in the storyboard and Control-drag from the yellow circle at the top to the Detail View Controller. Make it a **Present Modally** segue with the identifier set to **ShowDetail**.

The storyboard should look 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 need to tell the Detail screen to close when the landscape view is hidden.

► 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 Console output you should see that the `DetailViewController` is properly deallocated when you rotate back to portrait.

► If you're happy with the way the code 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 this chapter under **40 – Refactoring** in the Source Code folder.

Chapter 41: Internationalization

So far, the apps you’ve made in this book 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 to “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.

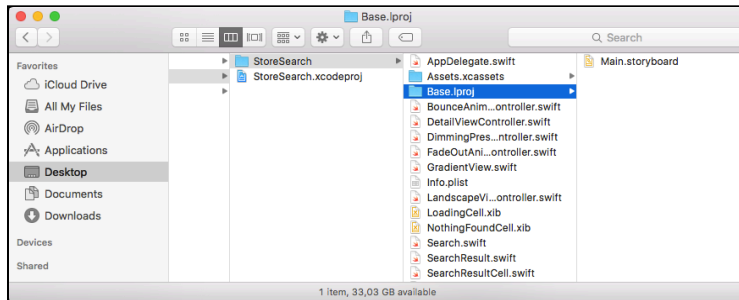
In this chapter, to get your feet wet with localization, you’ll add support for Dutch. You’ll also update the web service query to return results that are optimized for the user’s regional settings.

You’ll cover the following items:

- **Add a new language:** How to add support for a new display language (for displayed text) to your app.
- **Localize on-screen text:** How to localize text values used in code.
- **InfoPlist.strings:** Localize Info.plist file settings such as the app name.
- **Regional Settings:** Modify the web query to send the device language and region to get localized search results.

Add a new language

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 at least the storyboard, **Main.storyboard**. The Base.lproj folder is for files that can be localized. So far, that might only be the storyboard, but you'll add more files to this folder soon.

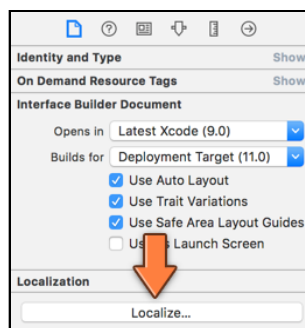
When you add support for another language, a new **XX.lproj** folder is created with XX being the two-letter code for that new language (**en** for English, **nl** for Dutch).

Localize a nib file

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).

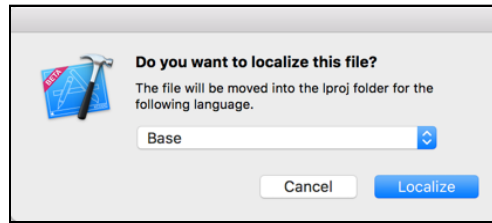
Because the NothingFoundCell.xib file isn't in any XX.lproj folders, it does not have any localizations yet.



The NothingFoundCell has no localizations

- Click the **Localize...** button in the Localization section.

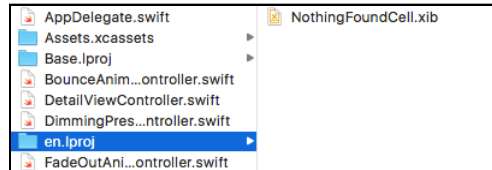
Xcode asks for confirmation because this involves moving the file to a new folder:



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 been moved to 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.

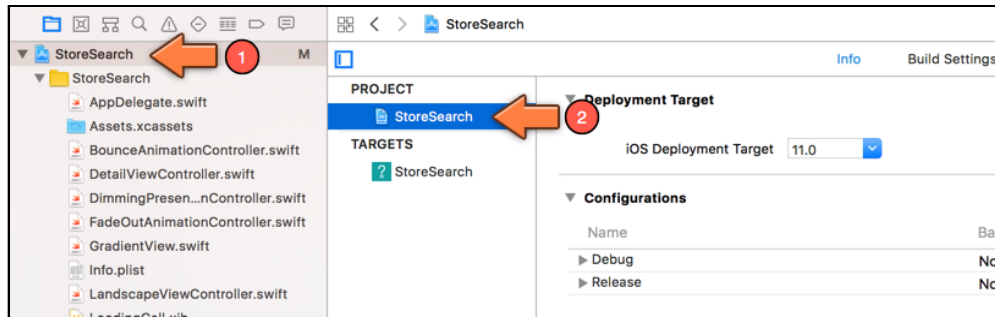


The Localization section now contains an entry for English

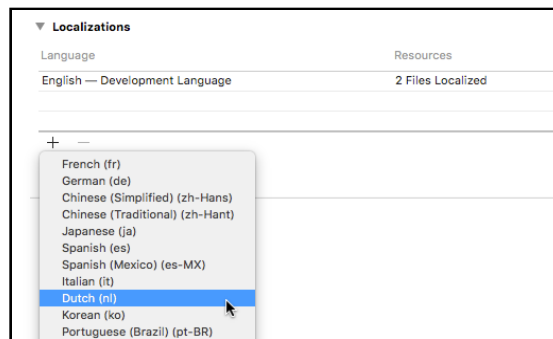
Add support for a new language

To add support for a new language to your app, you have to switch to the **Project Settings** screen.

- Click on **StoreSearch** at the top of the Project navigator to open the settings page. From the central sidebar, choose **StoreSearch** under **PROJECT** (not under TARGETS). (If the central sidebar isn't visible, click the small blue icon at the top of the sidebar area to open it.)

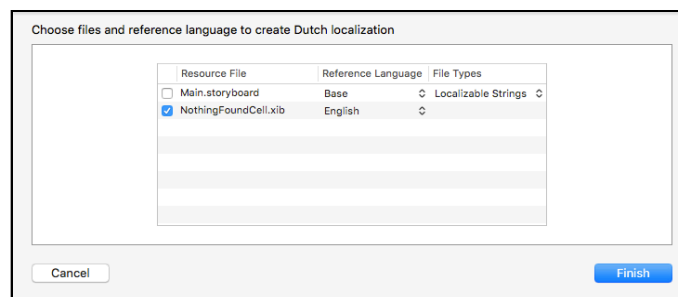
*The Project Settings*

► 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:

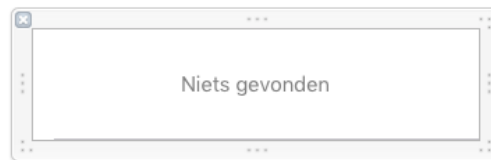


NothingFoundCell.xib has two localizations

Edit a language specific nib

Let's edit the Dutch version of this nib.

- Click on **NothingFoundCell.xib (Dutch)** to open it in Interface Builder.
- Change the label text to **Niets gevonden**.



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.

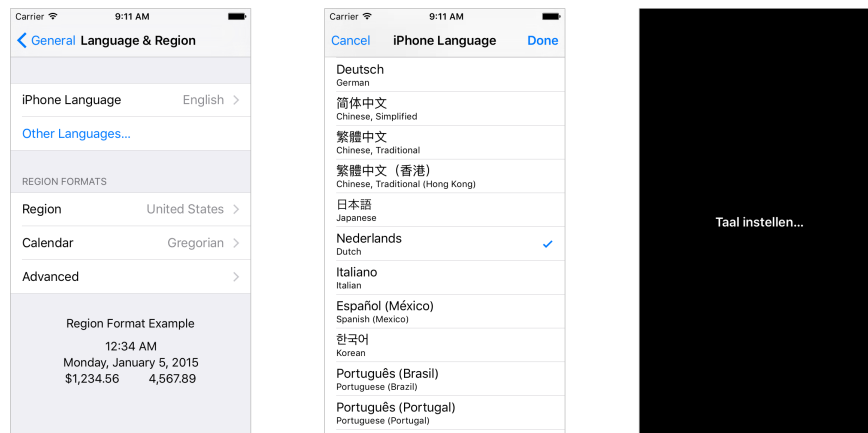
If you run the app now, nothing will have changed. You have to switch the Simulator to use the Dutch language first. However, before you do that, you really should remove the app from the simulator, clean the project, and do a fresh build.

The reason for this is that the nibs were previously not localized. If you were to switch the simulator's language now, the app might still use the old, non-localized versions of the nibs, or it might not. It's better to be safe than tear your hair out wondering what went wrong, right?

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.

Switch device language

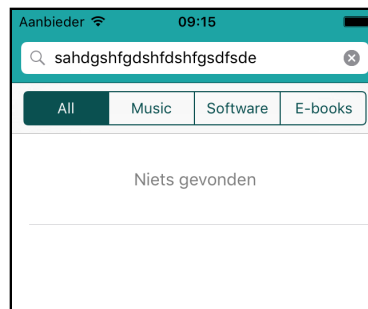
- Remove the app from the Simulator. Do a clean (**Product** → **Clean** or **Shift-⌘-K**) and re-build the app.
- Open the **Settings** app in the Simulator and go to **General** → **Language & Region** → **iPhone Language**. From the list pick **Nederlands (Dutch)**.



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**.)

Base internationalization

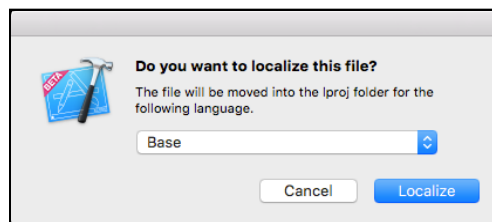
To localize the other nibs, you could repeat the process and add copies of their xib files to the **nl.lproj** folder. That isn't too bad for this app, but if you have an app with really complicated screens, then having multiple copies of the same nib can become a maintenance nightmare.

Whenever you need to change something on that screen, you need to update all of those nibs. There's a risk that you might overlook one or more nib files and they'll be 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.

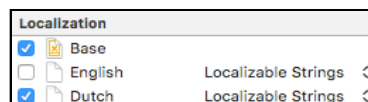
► Open **LoadingCell.xib** in Interface Builder. In the **File inspector** press the **Localize...** button. This time use **Base** as the language:



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**:

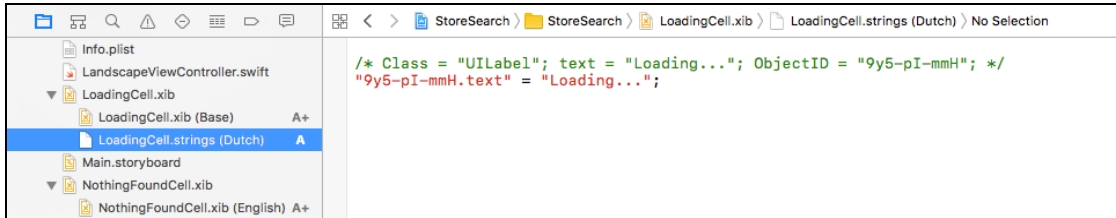


Adding a Dutch localization

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 disclosure triangle 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:

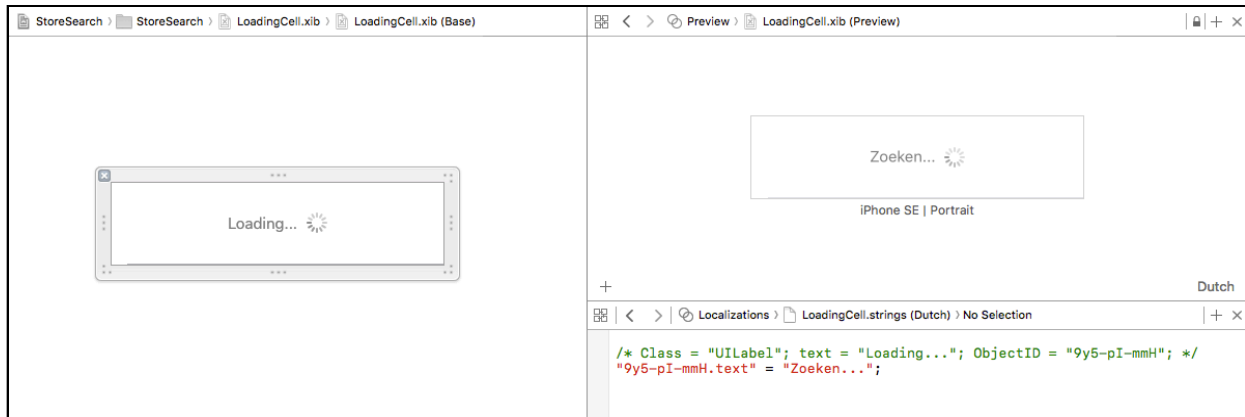
```
/* Class = "UILabel"; text = "Loading..."; ObjectID = "hU7-Dc-hSi"; */
"hU7-Dc-hSi.text" = "Loading...";
```

The green bit is a comment, just like in Swift. The second line says that the **text** property of the object with ID “hU7-Dc-hSi” 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...**

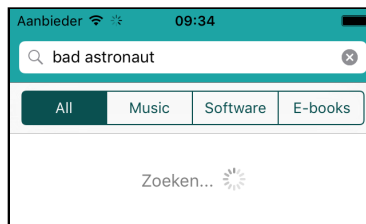
Tip: You can use the Assistant editor in Interface Builder to get a preview of your localized nib. Go to **LoadingCell.xib (Base)** and open the Assistant editor. From the Jump bar at the top, choose **Preview**. In the bottom-right corner it says English. Click this to switch to a Dutch preview.



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** (to be safe) 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**.

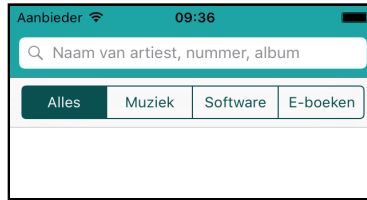
```
"68e-CH-NSs.placeholder" = "Naam van artiest, nummer, album";
```

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:**

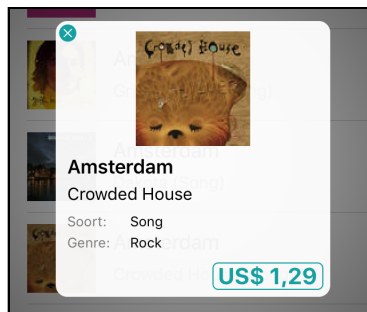
```
"DCQ-US-EVg.text" = "Soort:";
```

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.

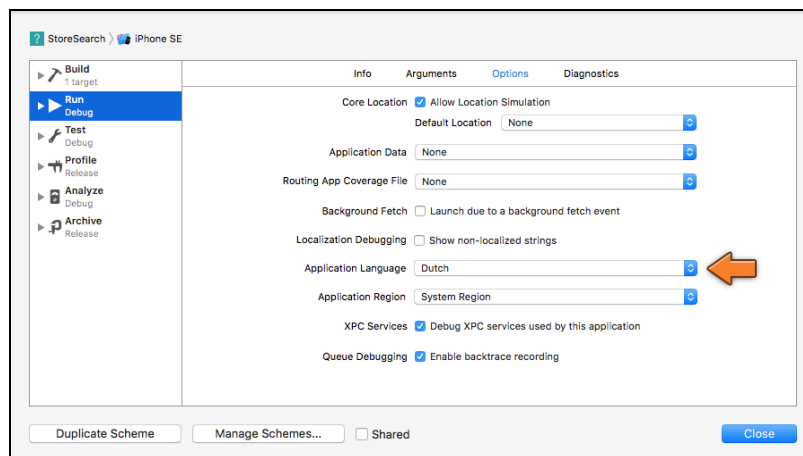
The Landscape View Controller doesn't have any text to translate.

► 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.

Localize on-screen text

Even though the nibs and storyboard have been translated, not all of the text is. For example, in the one-before-the-previous image 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 `type` property will be understood by non-English speaking users.

Localize text used in code

To localize text that is 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 type property with:

```
var type:String {
    let kind = self.kind ?? "audiobook"
    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 "(.)"** and press **return** to search.

In the replacement box type:

return NSLocalizedString("\$1", comment: "Localized kind: \$1")

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.

Make sure only the relevant search results from **SearchResult.swift** are selected – you don't want to make this change to all of the search results! Click **Replace** to finish.

Thanks to Scott Gardner for the tip!

The structure of type 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 text values as before because you haven't translated anything yet.

First, you need to create an empty `Localizable.strings` file.

➤ Right click on the yellow **StoreSearch** folder in the Project navigator, **New File...**, select the **Strings File** template under **iOS - Resources** and tap **Next**. Save the file as **Localizable.strings**.

➤ Select the **Localizable.strings**, in the File inspector (on the right) click **Localize...**, select **English** from the dropdown, and click **Localize**.

This creates an empty English **Localizable.strings** file. You need to use a command line tool named **genstrings** to populate the file with text strings from your source files. This requires a trip to the Terminal.

Generate localizable text strings

► 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 the text strings from those source files to the **Localizable.strings** file in the **en.lproj** folder.

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.

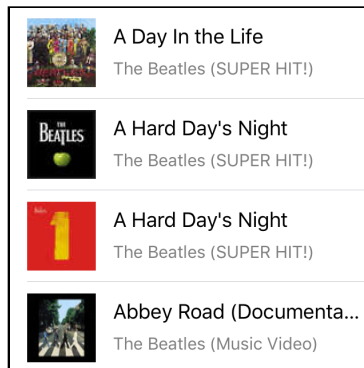
Tip: It's a good idea to make these comments as detailed as you can. In the words of fellow tutorial author Scott Gardner:

"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 of "Song".



Where it used to say *Song* it now says *SUPER HIT!*

Of course, changing the text in the English localization doesn't make much sense - reverse the change to `Song` 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 you 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() to all the text strings 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"  
  
// LandscapeViewController, showNothingFoundLabel()  
label.text = "Nothing Found"  
  
// SearchResultCell, configure(for)  
artistNameLabel.text = "Unknown"  
  
// SearchViewController, showNetworkError()  
title: "Whoops...",  
message: "There was an error reading from the iTunes Store.  
         Please try again.",  
title: "OK"
```

► Add NSLocalizedString() around these strings. 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 files to find the new strings. 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.

Localize dynamically constructed strings

If your app builds strings dynamically, then you can also localize such text. 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"),
    searchResult.artistName, searchResult.kindForDisplay())
```

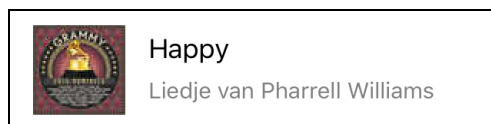
After running **genstrings** again, this shows up in Localizable.strings as:

```
/* Format for artist name */
"%@ (%@)" = "%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 instance 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())
```

Data-driven localization

There is one more thing I'd like to improve. Remember how in **SearchResult.swift** the type property is this enormous switch statement? That's "smelly" to me. The problem is that any new products require you to add 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 type as it's almost identical):

```
private let typeForKind = [
    "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 type becomes really short:

```

var type: String {
    let kind = self.kind ?? "audiobook"
    return typeForKind[kind] ?? kind
}

```

It's nothing more than a simple dictionary lookup.

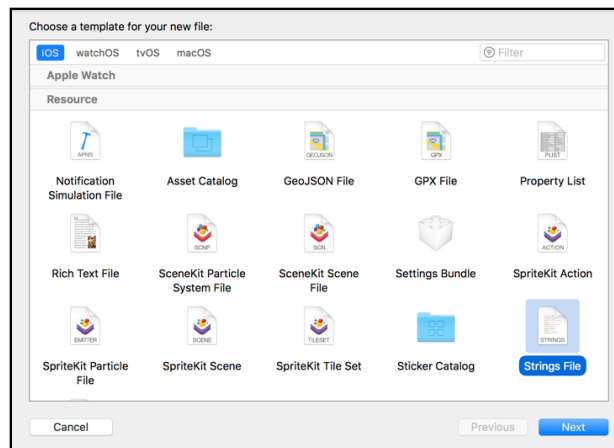
The ?? is the nil coalescing operator. Remember that dictionary lookups always return an optional, just in case the key you're looking for – kind in this case – does not exist in the dictionary. That could happen if the iTunes web service added new product types. If the dictionary gives you nil, the ?? operator simply returns the original value of kind.

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 strings 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 **English** 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’s for 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.

Fix web request include language and region

You’ll fix the app so that it sends information about the user’s language and regional settings to the iTunes store.

► In **Search.swift**, change the `iTunesURL(searchText:category:)` method as follows:

```
private func iTunesURL(searchText: String,
                        category: Category) -> URL {
    // Add the following 3 lines
    let locale = Locale.autoupdatingCurrent
    let language = locale.identifier
    let countryCode = locale.regionCode ?? "en_US"
    . . .
    // Modify the URL string
    let urlString = "https://itunes.apple.com/search?" +
        "term=\(encodedText)&limit=200&entity=\(kind)" +
        "&lang=\(language)&country=\(countryCode)"

    let url = URL(string: urlString)
    print("URL: \(url!)" ) // Add this
    return url!
}
```

The regional settings are also referred to as the user's **locale** and of course there is an object to represent 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 their regional information while the app is running, the app will automatically use these new settings the next time it does something with the `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 `?? "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 something like the following (if you have English set as the language):

```
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).

Test for region changes

► 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**. If the language is not set to Dutch, then set the language to Dutch now.

► 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 are both now set 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, but it 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.

You can find the project files for this chapter under **41 – Internationalization** in the Source Code folder.

Chapter 42: The iPad

Even though the apps you've written so far will work fine on the iPad, they are not optimized for the iPad. There really isn't much difference between the iPhone and the iPad: they both run iOS and have access to the exact same frameworks. But the iPad has a much bigger screen (768×1024 points for the regular iPad, 834×1112 points for the 10.5-inch iPad Pro, 1024×1366 points for the 12.9-inch iPad Pro) and that makes all the difference.

Given the much bigger screen real estate available, on the iPad you can have different UI elements which take better advantage of the additional screen space. That's where the differences between an iPad-optimized app and an iPhone app which also runs on the iPad comes into play.

In this chapter you will cover the following:

- **Universal apps:** A brief explanation of universal apps and how to switch from universal mode to supporting a specific platform only.
- **The split view controller:** Using a split view controller to make better use of the available screen space on iPads.
- **Improve the detail pane:** Re-using the Detail screen from the iPhone version (with some adjustments) to display detail information on iPad.
- **Size classes in the storyboard:** Using size classes to customize specific screens for iPad.
- **Your own popover:** Create a menu popover to be displayed on the iPad.
- **Send e-mail from the app:** Send a support e-mail from within the app using the iOS e-mail functionality.

- **Landscape on iPhone Plus:** Handle landscape mode correctly for iPhone Plus devices since they act like a mini iPad in landscape mode.

Universal apps

Before Xcode 9 and iOS 11, when you created a new project, you could specify whether the app was going to be a *universal app* (which would run on both iPhone and iPad) or if it was going to support only one platform.

In Xcode 9, you don't have to make this choice - all apps are universal apps by default. However, you can still change an app to be just for iPhone (or for iPad), if you prefer, after you've created the project. You will *not* be doing that for *StoreSearch*, but in case you want to know how to change your app from a universal app to one which supports a particular platform. here's how you do it.

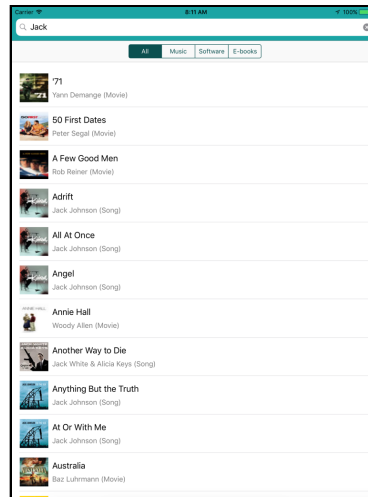
➤ Go to the **Project Settings** screen and select the *StoreSearch* target.

In the **General** tab under **Deployment Info** there is a setting for **Devices**. It should be set to **Universal** (and that's where you want it to be) but if you wanted to, you can change to one of the other values.



How to change device support

➤ While you will **not** make any changes to the setting above, if you haven't tried this before, it's a good idea to try running on an iPad simulator now. Be aware that the iPad Simulator is huge, so you may need to use the **Window** → **Scale** option from the Simulator menu to make it fit on your computer.



StoreSearch in the iPad Simulator

This works fine, but as I said before, simply blowing up the interface to iPad size does not take advantage of all the extra space the bigger screen offers. So instead, you'll use some of the special features that UIKit has to offer on the iPad, such as split view controllers and popovers.

The split view controller

On the iPhone, with a few exceptions such as when you embed view controllers inside another, a view controller generally manages the whole screen.

On the iPad, because the display is so much bigger, it is common for view controllers to manage just a section of the screen. 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 smaller pane on the left (the “master” pane) usually containing a list of items, and a larger right pane (the “detail” pane) showing 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 standard apps such as Mail and Settings.



The split view controller in landscape and portrait orientations

If the iPad is in landscape mode, 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/hide it.)

In this section, you'll convert the app to use a split view controller. This has some consequences for the organization of the user interface.

Check the iPad orientations

Because the iPad has different dimensions than 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 the window.

► **Open Info.plist.** There will be a **Supported interface orientations** item with three items under it, and a **Supported interface orientations (iPad)** item with four items under it.

▼ 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)

The supported device orientations in Info.plist

The iPad has its own supported orientations. On the iPhone, you usually don't want to enable Upside Down but on the iPad you do. If the settings do not correspond to the above, do make sure to change them to match the screenshot.

Next, 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.

Add a split view controller

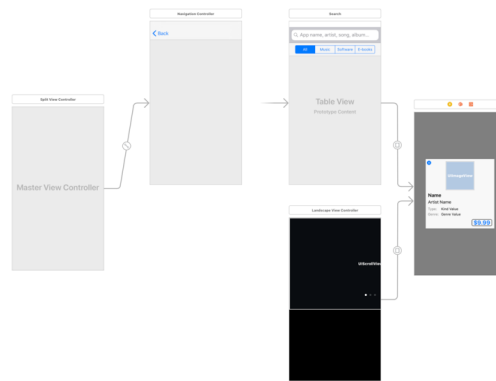
On the latest Xcode versions, you can simply add a Split View Controller object to the storyboard. The split view is only visible on the iPad; on the iPhone it stays hidden. This is a lot simpler than in previous iOS versions where you had to make two different storyboard files, one for the iPhone and one for the iPad. Now you just design your entire UI in a single storyboard and it magically works across all device types.

► Open **Main.storyboard**. If you are still in landscape mode, switch back to portrait mode now.

► Drag a new **Split View Controller** on to the canvas.

► The Split View Controller comes with several scenes pre-attached. Remove the white View Controller. Also remove the one that says Root View Controller. Keep just the Master View Controller and the Navigation Controller.

Here's the final result after I was done:



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.

► Control-drag from the Split View Controller to the Search scene. 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?

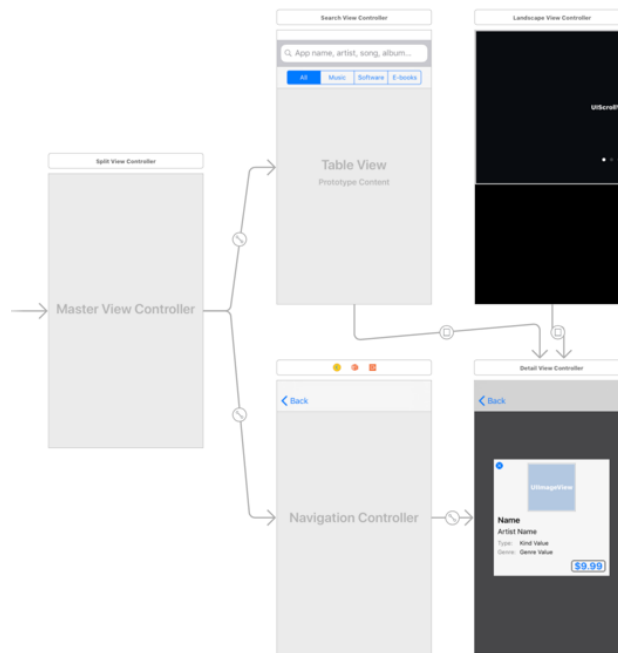
► Control-drag from the Split View Controller to the Navigation Controller. Choose **Relationship Segue – detail view controller**.

► Control-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.

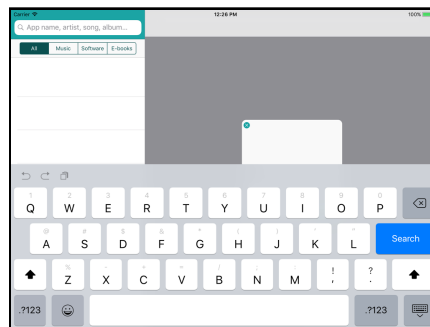
► Pick up the arrow (tap on the arrow to select it first and then drag) that points to the Search scene and drag it over to the Split View Controller. (You can also check the **Is Initial View Controller** option in the Attributes inspector for the Split View Controller.)

Now everything is connected:



The master and detail panes are connected to the split view

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...

You'll fix the app over the course of this chapter to make sure it doesn't do anything funny on the iPad!

Fix the master pane

The master pane 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 – what's known as the *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` – to 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 splitVC: UISplitViewController {  
    return window!.rootViewController as! UISplitViewController  
}  
  
var searchVC: SearchViewController {  
    return splitVC.viewControllers.first as! SearchViewController  
}  
  
var detailNavController: UINavigationController {  
    return splitVC.viewControllers.last as! UINavigationController  
}  
  
var detailVC: DetailViewController {  
    return detailNavController.topViewController  
        as! DetailViewController  
}
```

These four computed properties refer to the various view controllers in the app:

- `splitVC`: The top-level view controller.
- `searchVC`: The Search screen in the master pane of the split view.
- `detailNavController`: The `UINavigationController` in the detail pane of the split view.
- `detailVC`: 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 for the previous apps.

► Add the following line to `application(_:didFinishLaunchingWithOptions:)`:

```
detailVC.navigationItem.leftBarButtonItem =  
    splitVC.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 in portrait mode is a back arrow:



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 mode:



The display mode button has a title

Exercise. On the iPad, rotating 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.

Improve 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.

To pop-up or not to pop-up

► Add the following instance variable to **DetailViewController.swift**:

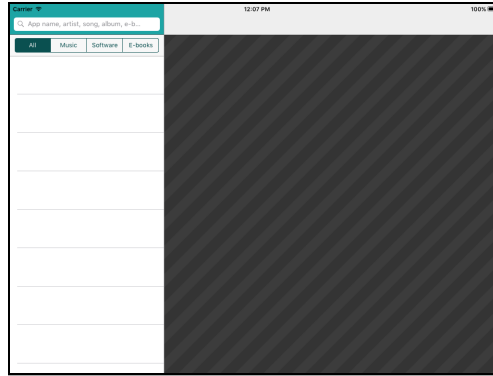
```
var isPopUp = false
```

► In `viewDidLoad()` replace the four lines dealing with the gesture recognizer set up and the one setting up the background color, with the following:

```
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  
}
```

With the gesture recognizer code inside the `if isPopUp` check, tapping the background has no effect on the iPad. Likewise for the line that sets the background color to `clearColor`.

The `else` branch always hides the pop-up view 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 need `SearchViewController` to tell the `DetailViewController` that a new `SearchResult` has been selected.

Previously, on an iPhone, `SearchViewController` created a new instance of `DetailViewController` every time you tapped a row, but now, on an iPad, 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.

► First, 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.

► Add the following line to `application(_:didFinishLaunchingWithOptions:)` in **`AppDelegate.swift`**:

```
searchVC.splitViewDetail = detailVC
```

► To change what happens when the user taps a search result on the iPad, replace `tableView(_:didSelectRowAt:)` in **`SearchViewController.swift`** 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 is running on an iPhone, you look 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 iPhone 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`. So, let's make that happen.

The ideal place to update the labels 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.

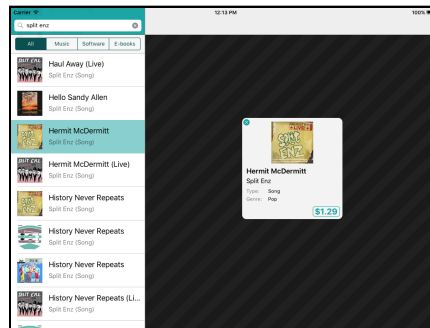
Notice that you first check whether the controller's view is already loaded. It's possible that `searchResult` is given an object when the `DetailViewController` hasn't loaded its view yet – which is exactly what happens in the iPhone version of the app. In that case, you don't want to call `updateUI()` 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

Fix the Detail pop-up for iPhone

One small problem: the Detail pop-up no longer works properly on the iPhone because `isPopUp` is always false (try it out...).

- 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.

Display the app name on Detail pane

It would be nice if the app showed its name in the navigation bar on above 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 happens, 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`.

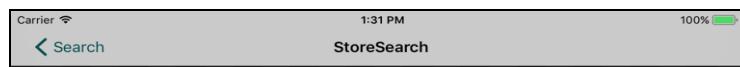
Because `NSBundle`'s `localizedInfoDictionary` can be `nil` you need to unwrap it. The value stored under the `"CFBundleDisplayName"` key may also be `nil`. And finally, the `as?` cast to turn the value to a `String` can also potentially fail. If you're counting along, that is three things that can go wrong in this single 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 set to 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 (English)**:

```
CFBundleDisplayName = "StoreSearch";
```



That's a good-looking title

There are a few other small improvements to make.

Remove input focus on iPad

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 `viewDidLoad()` in **SearchViewController.swift**, enclose the call to `becomeFirstResponder()` in a condition:

```
if UIDevice.current.userInterfaceIdiom != .pad {  
    searchBar.becomeFirstResponder()  
}
```

To figure out whether the app is running on an iPhone or on an iPad, you look at the current `userInterfaceIdiom`. This is either `.pad` or `.phone` – an iPod touch counts as a phone in this case.

Hide the master pane in portrait mode

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**:

```
private 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 after the `if case .results` block:

```
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 away 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.

If you're lucky, you may be able to use the same view controllers for both versions of the app but often, you might find that the iPad user interface for your app is different enough from the iPhone's that you have to make all new view controllers with some duplicated logic.

The Apple Developer Forums

When I first 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”!

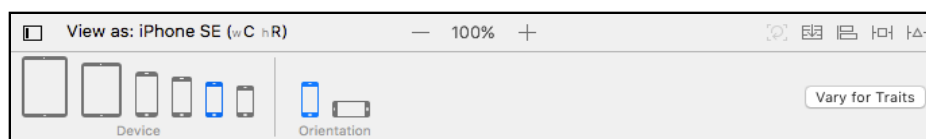
So if you're stuck, don't forget to look at the Apple Developer Forums for a solution: <https://forums.developer.apple.com>

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 space on an iPad 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 there is a better way. This is exactly the sort of thing size classes were invented for!

► Open **Main.storyboard** and take a look at the **View as:** pane.

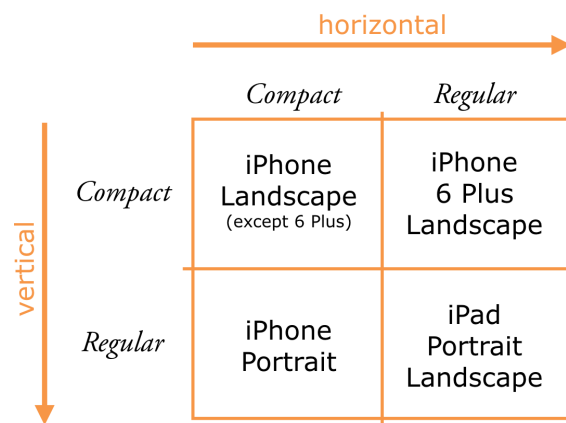


Size classes in the View as: pane

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

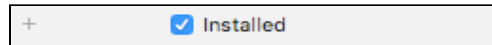
We want to make the Detail pop-up bigger when the app runs on the iPad. However, if you make any edits to the storyboard right now, these edits will also affect the design of the app in iPhone mode. Fortunately, there is a way to make edits that apply to a specific size class only.

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.

Uninstall an item for a specific size class

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.

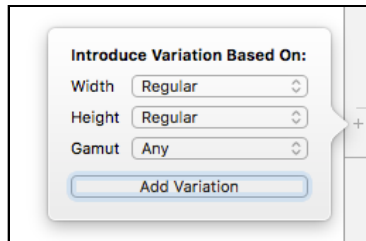
► Select the **Close Button**. Go to the Attributes inspector and scroll all the way to the bottom, to the **Installed** option.



The installed checkbox

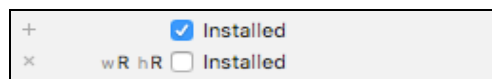
This option lets you remove a view from a specific size class, while leaving it visible in other size classes.

► 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

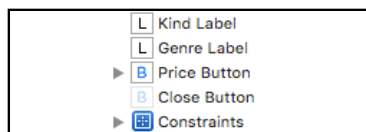
This adds a new line with a second Installed checkbox:



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 still see the button in the Document Outline, but it is grayed out:

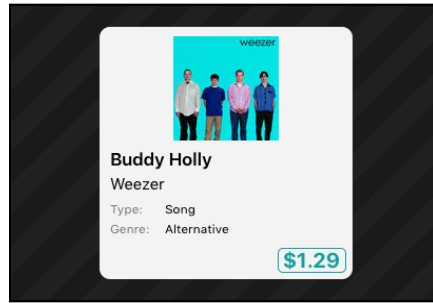


The Close Button is still present but grayed out

► 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 size classes!

- Run the app and you'll see that the close button really is gone on the iPad:

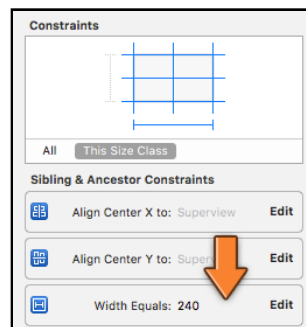


No more close button in the top-left corner

Change the storyboard layout for a given size class

Using the same principle as above, you can change the layout of the Detail screen to be completely different between the iPhone and iPad versions. For example, you can change the Detail pop-up to be bigger on an iPad.

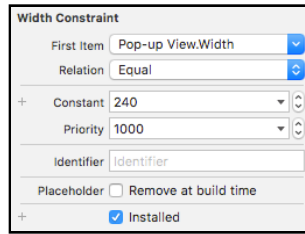
- In the storyboard, switch to the **iPad Pro** layout again.
- Select the **Pop-up View** and go to the **Size inspector**. The **Constraints** section shows the constraints for this view:



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. So, do the following.

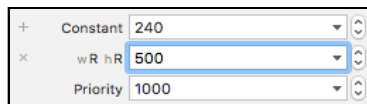
► Double-click **Width Equals: 240**. This brings up the **Size inspector** for just that constraint:



The Size inspector for the Width constraint

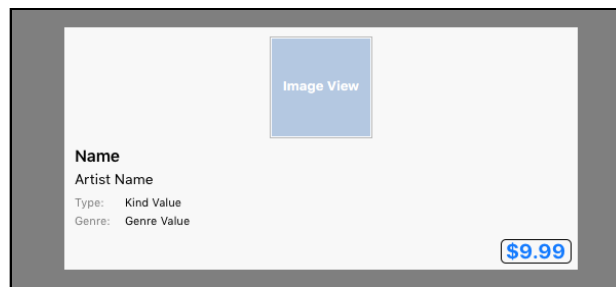
(If you just type in a new value for Constant, the constraint will become larger for all size classes again.)

► 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 the same way, change the **Width** and **Height** constraints of the **Image View** to **180**.

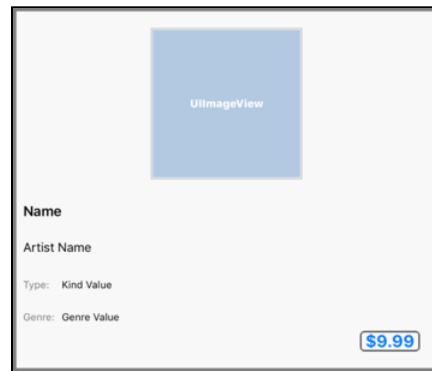
► Select the **Vertical Space** constraint between the **Name** label and the **Image View** and go to its **Size inspector**. Add a new variation for Constant and type **28** into the **wR hR** field.

► 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 the existing value.

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 sides instead of 8.

You should end up with something that looks like this:

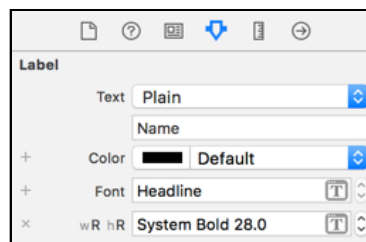


The Pop-up View after changing the vertical spaces

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.)

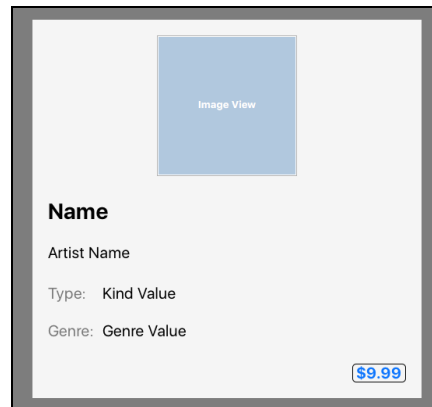
- Select the **Name** label. In the **Attributes inspector** click the + in front of **Font** to add a new variant. Choose the **System Bold** font, size **28**.



Adding a size class variation for the label's font

- Change the font of the other labels to **System**, size **20**. You can do this in one go by making a multiple-selection.
- Change all the “leading” **Horizontal Space** constraints to **20** for this size class.

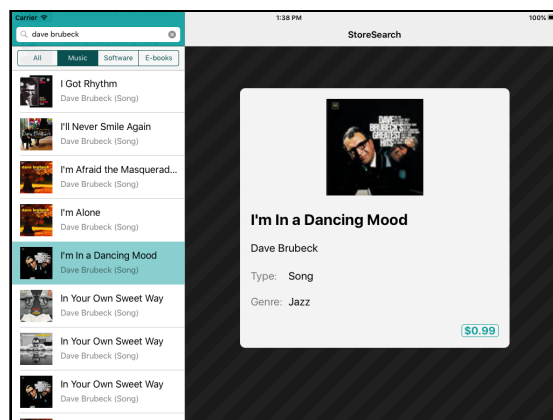
The final layout should look like this:



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 should work flawlessly.

Try it out: run the app on the iPad Air 2 or iPad Pro simulator. Swipe up from the bottom of the screen to have your dock appear on screen. Drag an app icon from the dock on to the right (or left) edge of the iPad screen and it should snap on, giving you a two apps running side-by-side. You can drag the divider bar to adjust the size occupied by each app. Thanks to size classes, the layout of *StoreSearch* will automatically adapt 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

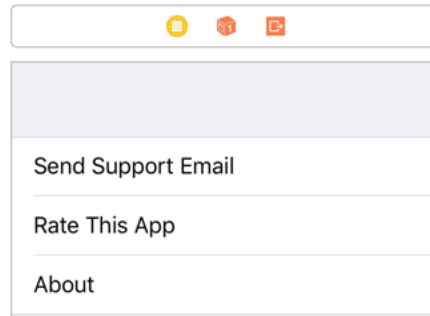
Anyone who has ever used an iPad before is no doubt familiar with popovers, the floating panels that appear when you tap a button in a navigation bar or toolbar. They are a very handy UI element.

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.

Add the menu items

- 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** on to 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**):



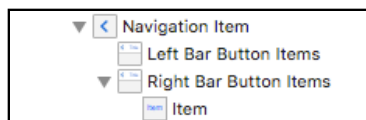
The design for the new table view controller

This just puts three items in the table. You will only do something with the first one in this book. Feel free to implement the functionality of the other two by yourself.

Display as popover

To show the new 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 Document Outline. Make sure the Bar Button Item is in the **Right Bar Button Items** group.

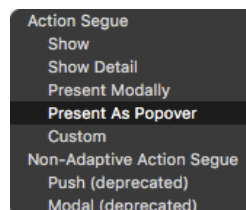


The new bar button item in the Navigation Item

- 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.

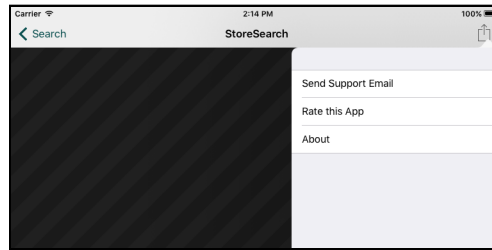
- Control-drag from the bar button (in the Document Outline) to the Table View Controller to make segue. Choose the segue type of **Action Segue – Present As Popover**.



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 should look like this:

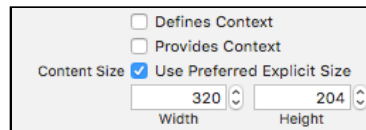


That menu is a bit too tall

Set the popover size

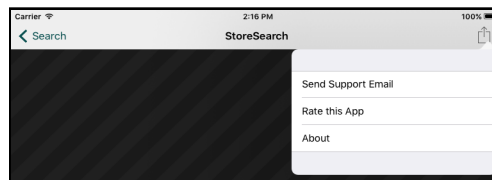
The popover doesn't really know how big its content view controller is, so it just picks a size. That's ugly, but you can tell it how big the view controller should be with the *preferred content size* property.

► 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

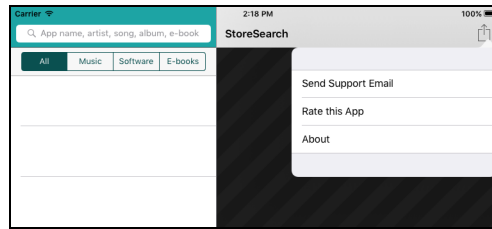
Now the size of the menu popover looks a lot more appropriate:



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 they 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, in portrait mode, the other bar button (Search) is still active as well. 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 (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 as to 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.

Do not allow two popovers to be open at the same time

The scenario you need to handle is to disable the user opening the master pane via a tap on the Search button, when in portrait mode, while they already have the menu popover open, or vice versa. To fix this issue, you need to know when the master pane becomes visible, so you can dismiss any other visible popover.

Wouldn't you know it... of course there is a delegate method for that.

► Add the following extension to the end 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 Console. 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:)`:

```
splitVC.delegate = self
```

And that should do it! Try having both the master pane and the popover open in portrait mode. Ten bucks says you can't!

Send e-mail from the app

Now, let's make the "Send Support Email" menu option work. Letting users send an E-mail 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 e-mail and then sends the e-mail 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 e-mail, 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.

The MenuViewController class

To make things 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 menuViewControllerSendEmail(_ 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:

```
// MARK:- Table View Delegates
override func tableView(_ tableView: UITableView,
                        didSelectRowAt indexPath: IndexPath) {
    tableView.deselectRow(at: indexPath, animated: true)

    if indexPath.row == 0 {
        delegate?.menuViewControllerSendEmail(self)
    }
}
```

Set the MenuViewController delegate

Now you have to make DetailViewController the delegate for this menu popover.

- Switch to **DetailViewController.swift** and add the following extension to the bottom of the source file to conform to the new protocol:

```
extension DetailViewController: MenuViewControllerDelegate {
    func menuViewControllerSendEmail(_: MenuViewController) {
    }
}
```

Currently, the code is just a stub - you'll fill in the implementation code in a bit.

- Next, add the following navigation code to the class:

```
// MARK:- Navigation
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 its delegate is.

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.

Show the mail compose view

► The `MFMailComposeViewController` lives in the `MessageUI` framework - import that in **DetailViewController.swift**:

```
import MessageUI
```

► Then, add the following code to `menuViewControllerSendEmail()` (in the extension at the end):

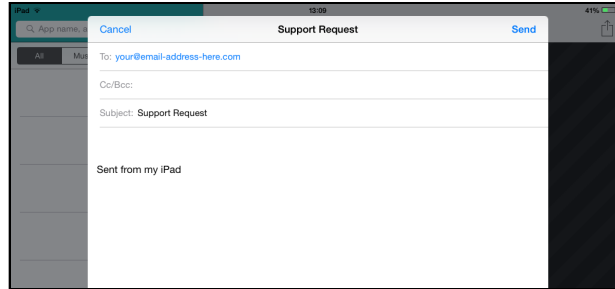
```
dismiss(animated: true) {  
    if MFMailComposeViewController.canSendMail() {  
        let controller = MFMailComposeViewController()  
        controller.setSubject(NSLocalizedString("Support Request",  
                                                comment: "Email subject"))  
        controller.setToRecipients(["your@email-address-here.com"])  
        self.present(controller, animated: true, completion: nil)  
    }  
}
```

The code first calls `dismiss(animated:)` to hide the popover. This method takes a completion closure that until now you've always left `nil`. Here you implement the 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 e-mail and the e-mail address of the recipient. You probably should put your own e-mail address there!

► Run the app and pick the Send Support Email menu option. The standard e-mail compose sheet should slide up.



The e-mail interface

Note: If you run the app on a device and don't see the e-mail sheet, you may not have set up any e-mail accounts on your device. It won't work on the Simulator at all though it did use to work for some previous iOS versions.

The mail compose view delegate

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 the mail composer view.

➤ 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 was 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 wanted. Check the documentation for the possible result codes.

➤ In the `menuViewControllerSendEmail()` method, add the following line (after the controller is created, of course):

```
controller.mailComposeDelegate = self
```

➤ Now, if you press Cancel or Send, the mail compose sheet gets dismissed.

Modal sheet presentation styles

Did you notice that the mail sheet did not take up the entire screen area in landscape, but when you rotate to portrait it does? That is called a **page sheet**.

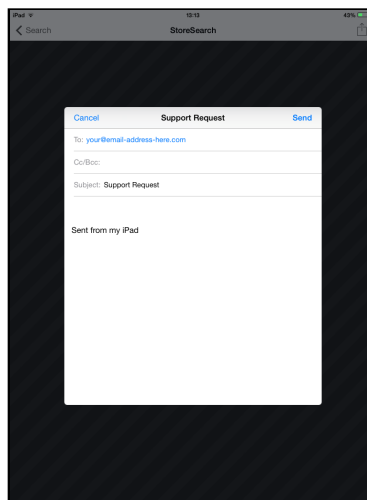
On the iPhone, if you presented a modal view controller, it always took over the entire screen, but on the iPad you have several options.

The page sheet is probably the nicest option for the `MFMailComposeViewController`, but let's experiment with the other ones as well, shall we?

► In `menuViewControllerSendEmail()`, 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

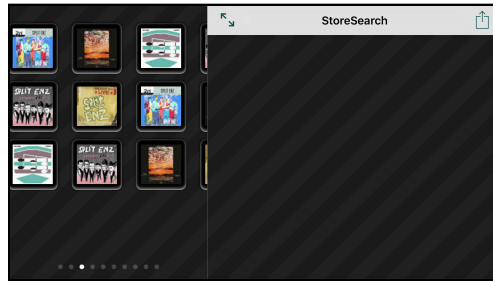
A form sheet is smaller than a page sheet, so it always takes up less room than the entire screen. There is also a “full screen” presentation style that always covers the entire screen, even in landscape. Try it out!

Landscape on iPhone 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 8 Plus** Simulator, do a search, and rotate to landscape.

The app will look something like this:



The landscape screen appears in the split-view's master pane

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 Plus devices are so big that they're almost small iPads. 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.)

Show split view correctly for iPhone Plus

To stop the LandscapeViewController from showing up, you have to make the rotation logic smarter.

► In **SearchViewController.swift**, change `willTransition(to:with:)` to:

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

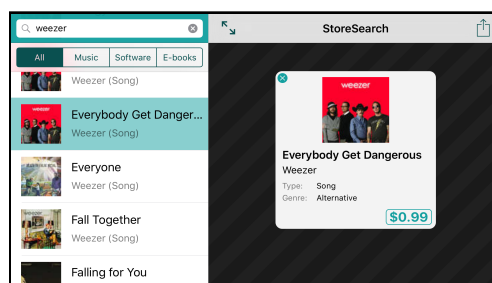
What about the size class? That sounds like it would be the obvious thing to tell the different devices apart. Unfortunately, looking at the size class *doesn't* work.

If the device is in portrait, the 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 Plus, you no longer show the landscape view, and you dismiss any Detail pop-up that may still be visible before you rotate to landscape.

► Try it out. Now the iPhone Plus shows a proper split view:



The app on the iPhone 8 Plus with a split-view

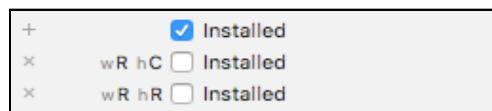
Add size class based UI changes for iPhone Plus

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 that size class, so the app uses the default design.

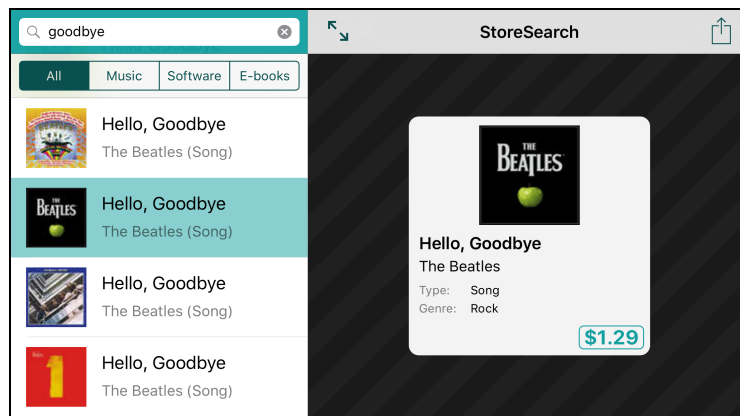
That's fine for the size of the Detail view, but it does mean the close button is visible again.

- Open the storyboard, open the **View as:** panel, switch to the **iPhone 8 Plus** mode and switch to landscape mode. (This will help you get the size classes right when you add exceptions.)
- Select the **Close Button** in the Detail scene. In the **Attributes inspector**, add a new row for **Installed** (for Width: Regular, Height: Compact) 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.



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 road.

- Celebrate by committing the final version of the source code and tagging it v1.0!

You can find the project files for this chapter under **42 – The iPad** in the Source Code folder.

Chapter 43: Distributing the App

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...)

Throughout this book, 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.

In this chapter, you'll learn how to beta test the *StoreSearch* app. After that, I'll also show you how to submit the app to the App Store, which is basically an extension of the same process. (By the way, I'd appreciate it if you don't actually submit the apps from this book. Let's not spam the App Store with dozens of identical *StoreSearch* or *Bull's Eye* apps.)

This chapter will cover the following:

- **Join the Apple Developer program:** How to sign up for the paid Apple Developer Program.
- **Beta testing:** How to beta test your app using Apple's TestFlight service.
- **Submit to the App Store:** How to submit your app to Apple for review before being made available on the App Store.

Join the Apple Developer 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 want to distribute apps within their own organization only. If you're still in school, the University Program may be worth looking into as well.

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, 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.

Beta testing

You will be distributing your app for beta testing via Apple's TestFlight service.

TestFlight

In the early days of iOS development, the only way to send builds to testers was via what was known as Ad Hoc distribution. You had to register specific devices for Ad Hoc distribution (for which you needed to know the unique ID for the device) and there was a limit of 100 devices per developer account. You could only reset the devices in this list once per year, when you renewed your developer account.

Additionally, you had to go through a complicated manual signing process to sign builds for Ad Hoc distribution and you had to send these builds out to your users and hope that they could figure out how to install the builds on their devices and

troubleshoot any installation issues by themselves, or provide you enough information to help them figure out what was going on.

All this changed with the introduction of Apple's TestFlight service.

TestFlight allows you to distribute your beta builds to 10,000 testers and all you need is just their e-mail address!

What's more, the process itself is fairly straightforward since you simply build your app in Xcode and upload to the App Store. The app binary would go through some processing at this point and once the processing is complete, you were able to offer the app for testing to your internal testers immediately.

The internal tester count is limited to 25 people though and they have to be part of your Apple Developer team. So, if you are a single developer, this probably would not work for you. But if you signed up for your developer account as an organization, then you can start testing immediately.

If you wanted to distribute to external testers (the 10,000 testers I mentioned earlier), you do have to go through an initial review of your app by Apple. This process is usually quite fast and takes about a day and this has to be done only once per new beta build - for the very first beta build. After that, you can simply push out new builds without needing to wait for approval from Apple.

To add new users to beta test, you simply invite them using their e-mail address. They will receive an invitation e-mail which they can accept (or reject by simply ignoring the e-mail). If they accept the invitation, they are prompted to install the TestFlight app which will handle installing beta builds and notifying users of updates to beta builds from then on.

You can read more on TestFlight at: <https://developer.apple.com/testflight/>

Apple Developer portal

While the new TestFlight workflow for beta testing is miles ahead of what you had previously, it still requires you to do a bunch of things on several different Apple sites. You start out on the Apple Developer portal where you need to create an App ID for your new app.

► Open your favorite web browser and navigate to the Developer Member Center at developer.apple.com/membercenter. Sign in and go to **Certificates, IDs & Profiles** on the left sidebar. (If you have trouble using the site and you are not using Safari, try using Safari. Other browsers can throw up weird issues sometimes.)

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.

- Click on **App IDs** under **Identifiers** in the sidebar. You should get a list of existing app IDs, press the + button on the top right to add a new App ID:

Creating a new App ID

- Fill in the **App ID Description** field. This can be anything you want – it's just for usage on the Provisioning Portal.
- The **App ID Prefix** field contains the ID for your team. You cannot modify this value.
- Under **App ID Suffix**, select **Explicit App ID**. In the **Bundle ID** field you must enter the identifier that you used when you created the Xcode project. For me that is **com.razeware.StoreSearch**.

The Bundle ID must match with the identifier from Xcode

If you want your app to support push notifications, In-App Purchases, or iCloud, then you can also configure that here. *StoreSearch* doesn't need any of that, so leave the other fields on the default settings.

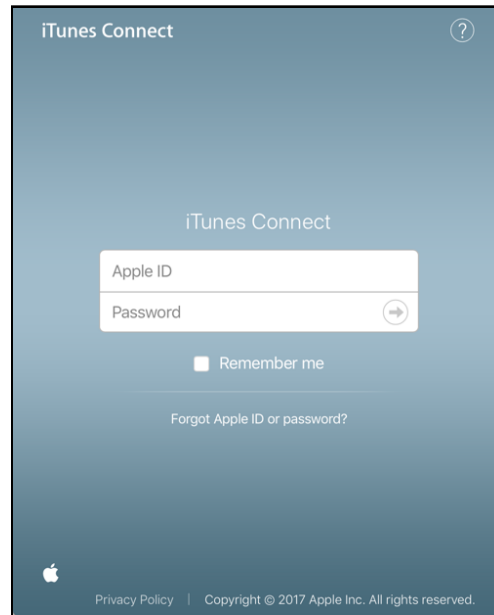
- Press **Continue** and then **Register** to create the App ID. The portal will now generate the App ID for you and add it to the list.

The full App ID is something like **U89ECKP4Y4.com.yourname.StoreSearch**. That number in front is your Apple Developer Team ID.

iTunes Connect

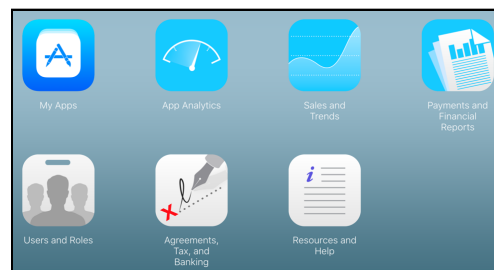
Next, you need to add your app to iTunes Connect.

- Navigate to itunesconnect.apple.com via your browser of choice. (Again, try using Safari if you run into any issues with the iTunes Connect site.)
- Log in using the same Apple ID that you used to sign up for your Apple Developer account.



Log in to iTunes Connect

- The first screen you see will look something like this. (If you are not an administrator for the iTunes Connect account, you might see fewer options on the screen than in the screenshot.)

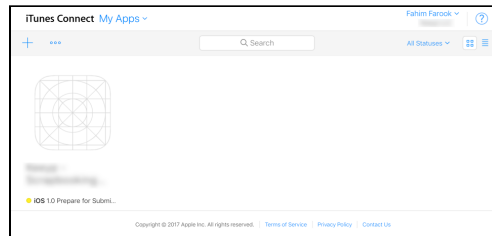


Initial iTunes Connect screen

- 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.

► Select **My Apps** - this is the option you need to manage everything related to your apps. You create new app entries, edit existing ones, and manage your beta testing and app distribution tasks all from there.

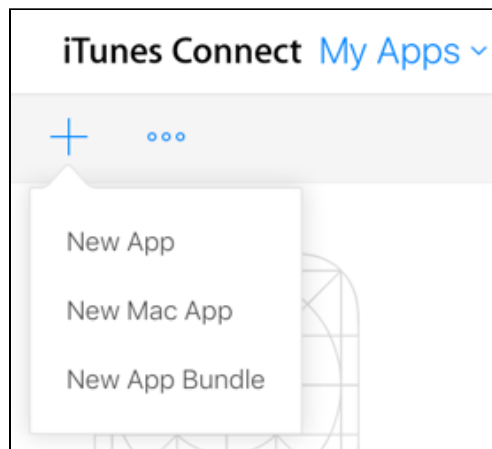
The page you are taken to lists your existing apps - if you have any. It also allows you to add new apps to iTunes Connect.



The My Apps page on iTunes Connect

Tip: If you are stuck and don't know what to do, you can use the help icon on the top right to access guides and videos which might allow you to figure out how to use the iTunes Connect site.

► Click the plus (+) icon on the top left to add a new app and then select **New App** from the menu.



Add a new app on iTunes Connect

► This should present you with a new dialog for entering the basic information necessary for an app.

New App

Platforms ?
☒ iOS ☐ tvOS

Name ?
StoreSearch

Primary Language ?
English (U.S.)

Bundle ID ?
Choose

Register a new bundle ID

SKU ?

Limit User Access (optional) ?
All App Managers, Developers, Marketers, and S...

Admin, Legal, Finance and Technical roles have access to all apps.

Cancel Create

New app information on iTunes Connect

Select the checkbox for **iOS** (since yours is an iOS app), enter the name of your app, select the primary language from the dropdown and select the Bundle ID from the dropdown.

The Bundle ID would be the app ID you added on the Apple Developer Portal earlier. If you don't see the app ID you added, try refreshing your browser or waiting for a bit in case the information has not updated on the Apple servers. Generally, the information should be reflected almost immediately.

Tip: If you are not sure about what you are supposed to enter for a particular field, you can always click on the question mark icon next to each field to get a hint as to what you should enter. Also note the hint in the above screenshot - you **must** have a bundle ID matching the ID you have in Xcode for your project. Otherwise, the upload from Xcode will fail.

The last value you have to enter is the **SKU** (or “skew”), which stands for **Stock-Keeping Unit**. This one confuses people a bit since it can be any unique value *for your company*. Basically, Apple does not care what this value is - it's only used for reporting purposes but it has to be unique for your apps. So, for example, if you use 1001 as the SKU for your first app, you *can't* use 1001 as the SKU for your second app too.

➤ Once you've filled in all the information, click **Create** and your new app will be added to iTunes - if there are no errors.

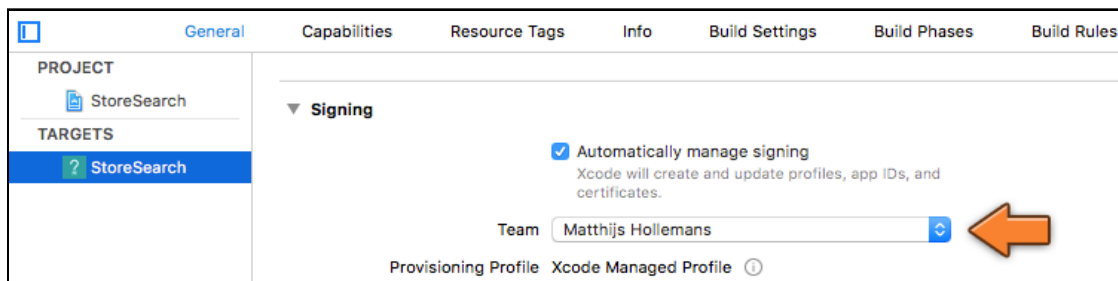
If your app name has been used before for another app (by anybody, not just you), or if your Bundle ID is not unique, or if your SKU has been used before, you will get an error message at this point. You would have to fix these issues and try again if this happens. Generally, it's the app name which gives you problems - so always a good idea to figure out if the name you selected is in use before you try to add a new app to iTunes Connect.

That's all you need to do at the iTunes end for the time being.

Upload for beta testing

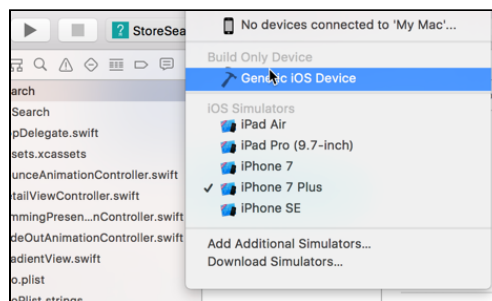
Once you have your app on iTunes Connect, you can upload the app for beta testing (and later submission to Apple) quite easily.

- Start Xcode, if you're not already running it, and then open the *StoreSearch* project.
- In the **Project Settings** screen, in the **General** tab, choose the correct **Team**. (As you noticed when you created the App ID, the team ID is connected to your App ID. So make sure that you have the right team selected here, otherwise you will run into issues later when you try to upload the build ...)



Choosing the team

- Change the device in the active scheme selector, on the Xcode toolbar, to **Generic iOS Device**.

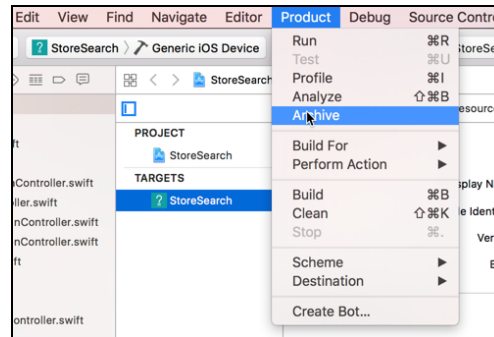


Selecting Generic iOS Device

Normally, when you build your app, you build for a specific Simulator or for a connected device because your intention at that point is to run the build on that particular Simulator or device.

But when you build for distribution, you have no idea which particular device a build would run on - so you have to build using the generic device setting in order to ensure that the resulting app would be compiled correctly to run on all supported devices.

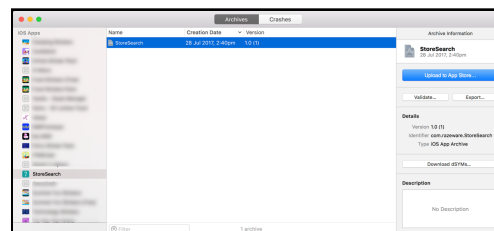
➤ Select **Product** → **Archive** from the Xcode menu.



Create app archive

If the Archive option is disabled, then you probably did not select the Generic iOS Device from the active scheme selector as per the previous step. You can only build an archive if you have the Generic iOS Device selected.

The app will compile the project and link it. If everything goes smoothly, Xcode should open the Organizer window and display the new archive which was just created.



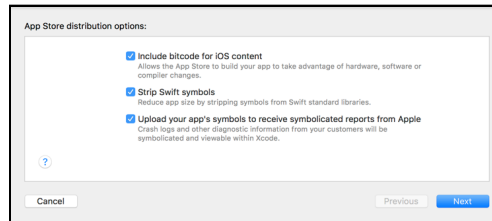
The Organizer window

You are now ready to upload the build to the App Store, as the big blue button on the right sidebar testifies :]

You can simply click the big blue button, or, you can click the **Validate...** button below it to verify that your app passes all of Apple's initial validations. The validations are run even if you use the Upload to App Store... button but the Validate... button is an easy way to check your button locally and verify that it passes muster before you upload it to the Apple servers.

► Click the **Upload to App Store...** button.

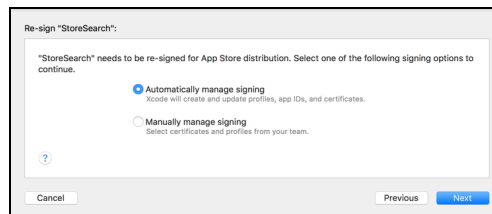
You will be asked to select some distribution options:



App Store distribution options

Each of the options has some helpful text describing what the option does - so you can basically go with the options that are suitable for you. If you are not sure, there should not be any harm in keeping all of the options checked.

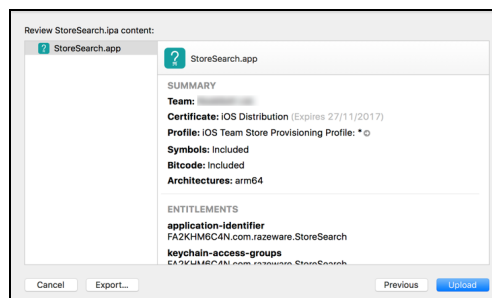
Tapping **Next** will take you to another dialog:



Code signing options

► Generally, it's best to go with **Automatically manage signing** unless you know what you are doing. The manual option gives you a lot more flexibility, but you also have to deal with the complexity that comes along with it :]

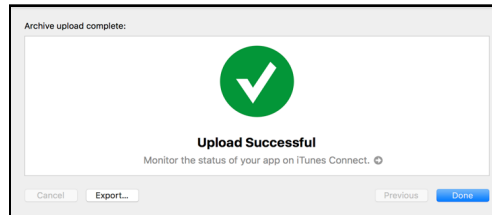
Tap **Next** to proceed and Xcode will work for a while signing your app and getting it ready for upload. When Xcode is ready, it should show you a screen similar to the following:



Ready to upload app

► Click **Upload** and Xcode will start uploading the binary for your app to the Apple servers. Depending on the size of your app (and the speed of your network connection), this might take a bit of time.

While the upload is in progress, you'll get a progress indicator and status messages indicating what is going on. If the upload completes successfully, you should get a message similar to the following:



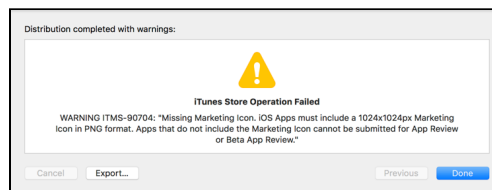
iTunes Connect app upload successful

If the upload completes successfully, that's all you have to do at the Xcode end. Sometimes though, you might get an error.

Some of these errors are basically incomprehensible since they might just be an iTunes error code and a cryptic message. If you are unlucky enough to get one of those, you might have to Google for the error code and see if you find somebody else who has figured out the issue. Usually, it turns out to be an Xcode issue or an iTunes issue that is resolved by Apple a few days later.

On the other hand, you might get specific error messages such as your app missing the app icon, or an app icon for a specific size that iTunes Connect expected. In such cases, fixing the issue by adding the missing assets and then creating another archive (you can't use the previous one) and uploading that should resolve the issue.

Sometimes, you might also get warnings, like the following:



iTunes Connect error about specific issue

The above indicates that the currently uploaded build is fine as far as passing the general validation but that it's missing an icon. While this is just a warning, as the message indicates, you still cannot use this build for external beta testing or for submitting to the App Store for review. So you'll need to fix the issue and upload a new new build.

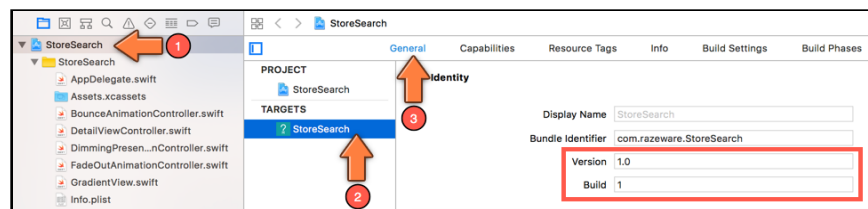
One thing you have to watch out for when uploading several builds for the same app to the Apple servers is the build number ...

Build number

Each build you submit to Apple has to be uniquely identifiable. How this is generally done for Xcode projects is by combining the version number and build number for the project to get a unique value.

But where is this version number and build number, you ask? Easy enough.

In Xcode, go to your project root in the Project navigator, select your project target, go to the **General** tab, and then check the **Identity** section.

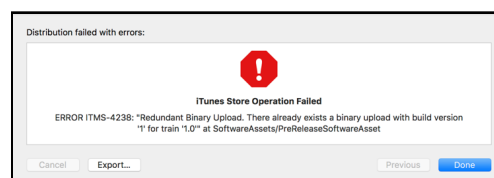


Xcode version and build numbers

For each new version of your app, you need to increment your version number. For each build you submit for a version, the build number has to change too, but, while you cannot repeat a version number, you can repeat a build number as long as the same build number is not used within a given version.

So, for example, you can have the builds for version 1.0 start from 1 and go up to however many builds as you like. And while you cannot use build number 1 for another build for version 1.0, if you start a new version, say 2.0, then you can start the build numbers for the new version again from 1 and go up incrementally.

So, if you've already uploaded a build to iTunes Connect and you have to upload another build (either because of an error or because you made a code change), then you need to remember to change the build number. If you don't, you'd get an error during the upload process.



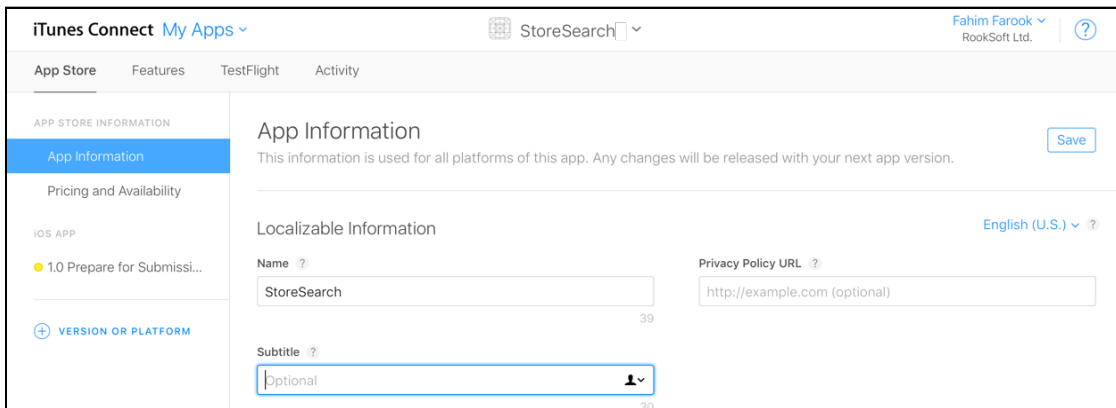
iTunes Connect error about build number

Change the build number in your project settings, create a new archive, upload it and you should be good to go!

Check your upload

You can check on the status of your uploaded build by logging into iTunes Connect.

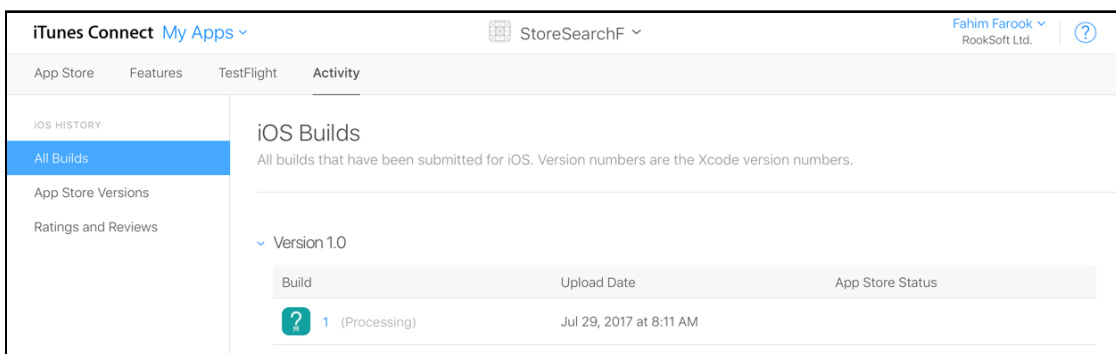
- Select **My Apps** from the main dashboard to get a listing of your apps and then select the *StoreSearch* app from there.
- You'll be taken to the app detail screen on iTunes Connect:



The app details on iTunes Connect

This is the screen where you would manage everything to do with a given app. You can edit the app information, add screenshots, change the price information, submit builds for review, and check the status of a build.

- Click on the **Activity** tab at the top. (The second row of text from the top.) This should take you to the latest activity for this particular app.



The activity page on iTunes Connect

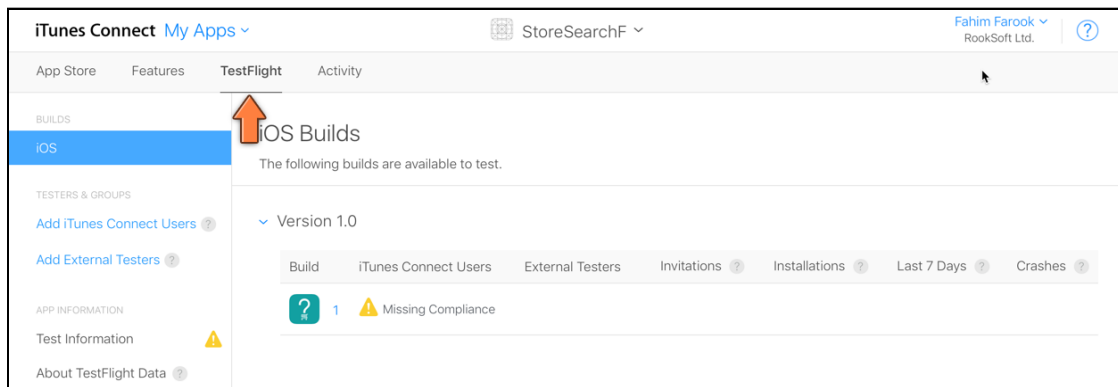
As the screen shows, your app is still processing. This can sometimes take a bit of time, though most of the time, the process is pretty quick. Once the processing is complete,

you should receive a notification e-mail from Apple indicating that your app had completed processing and is now available for either testing or distribution.

Internal testing

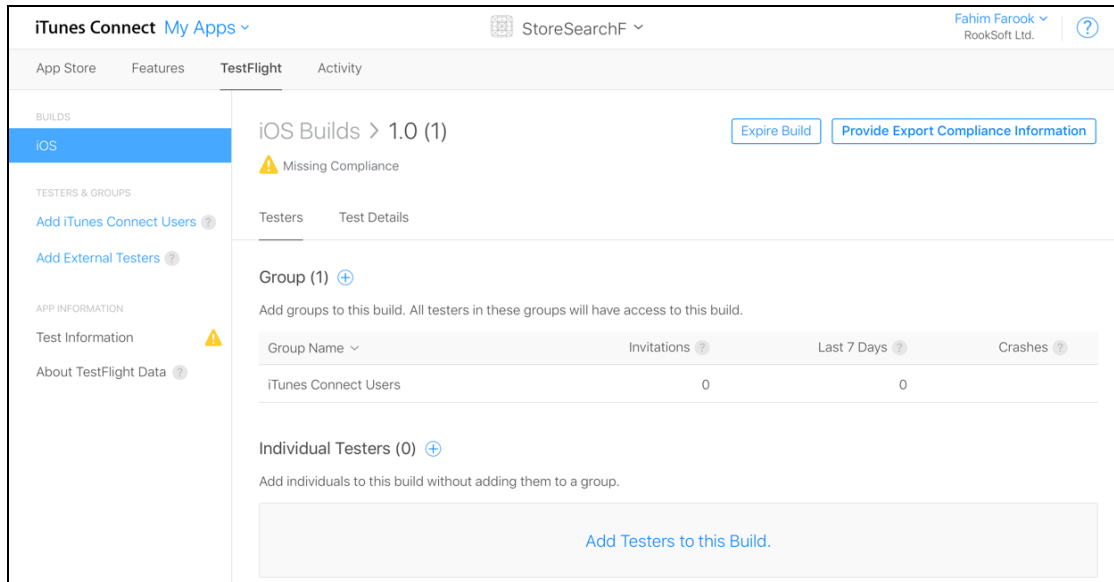
As I mentioned before, there are two test modes for TestFlight - internal and external. Once your app upload completes processing, you can immediately start internal testing.

► Log in to iTunes Connect and go to your app's detail screen. From there, select **TestFlight** from the list of tabs at the top.



The TestFlight page on iTunes Connect

As you'll notice, your app build has a warning - it's apparently missing compliance information. This is standard for all builds unless you provide the compliance information beforehand. The easiest way to fix this is to click on the blue "1" next to the app icon - that is a link which would take you to the detail information page for this particular build (build #1).



The build detail page on iTunes Connect

There, at the top of the page, is a big **Provide Export Compliance Information** button! Click on the button, go through a couple of screens of questions and you should be done with the export compliance stuff till you upload another build :]

The above screen also has a place to add testers for this build. So, you might think that is where you add testers for your app. Well ... yes, and no.

To add *internal* testers for your app, you actually have to go to the **Add iTunes Connect Users** link on the left of that screen. That takes you to a new screen from where you can select up to 25 existing users in your team to be added as internal testers.

If you don't have any team members at the moment though, you would need to first go back to your iTunes Connect dashboard, select the **Users and Roles** option, add some team members (and optionally, assign them the role of tester) and then come back and add the internal testers for your app.

The selected team members will be notified via e-mail that a new build is ready for testing and they will be asked to install the TestFlight iOS app so that they can participate in the testing process.

Internal testing, as the name implies, is generally for testing within your team. So Apple does not require your beta build to go through any sort of review before you start testing. But internal testing is also limited to just 25 people at most.

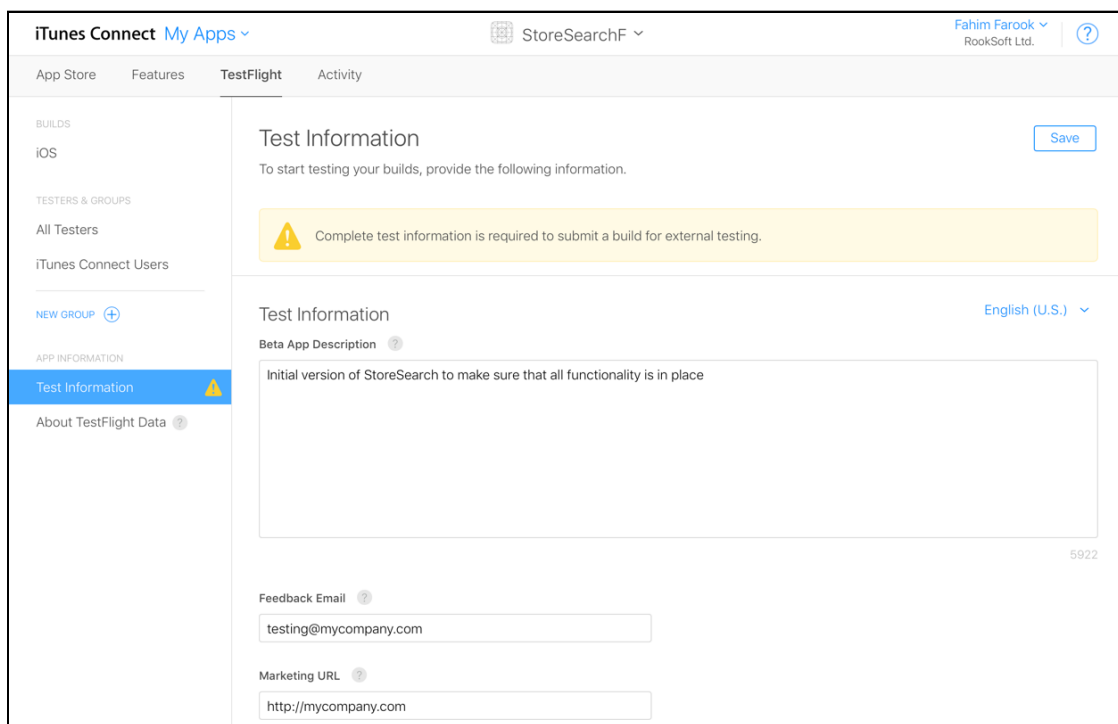
If you want to do more extensive beta testing, then you have to opt for external testing too.

External testing

External testing allows you to distribute beta builds of your app to up to 10,000 testers. But before you can start inviting testers, you have to get your beta build approved by Apple.

Before you can submit your beta build to Apple for review, you have to fill in the relevant test information for the particular build you want tested.

► Go to iTunes Connect, select your app, and on the app detail screen, go to the **TestFlight** tab. There should be an item named **Test information** on the left sidebar. Select it.



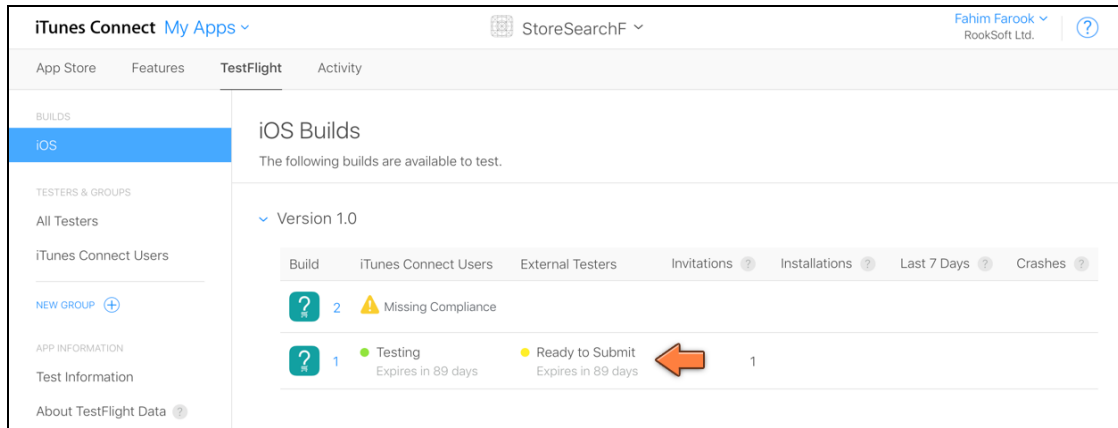
The screenshot shows the iTunes Connect interface for the app 'StoreSearchF'. The 'TestFlight' tab is selected, and the 'Test Information' section is active in the left sidebar. The main content area displays a yellow warning banner stating 'Complete test information is required to submit a build for external testing.' Below this, the 'Test Information' section includes a 'Beta App Description' field with the text 'Initial version of StoreSearch to make sure that all functionality is in place', a 'Feedback Email' field with 'testing@mycompany.com', and a 'Marketing URL' field with 'http://mycompany.com'. A 'Save' button is located in the top right corner of the form area.

Enter test information on iTunes Connect

Fill in at least the Beta App Description, Feedback Email, and Marketing URL values and the Beta App Review Information towards the end of the page and click **Save**.

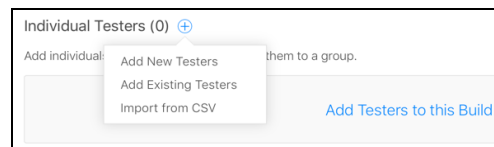
Tip: Don't forget to fill in the **Beta App Review Information** section towards the end. This section will remain incomplete till you enter your contact information towards the end of the page.

► Go to the **Builds - iOS** item on the left sidebar. Your build should now appear as "Ready to Submit".



The build is ready for beta review

- Go into the build details screen by clicking the blue "1" next to the app icon.
- Before you can submit the app for beta review, you need to add at least one external beta tester. So, use the **Add Testers to this Build** link towards the bottom of the screen or the blue plus (+) icon above it to start adding some external testers. Use the **Add New Testers** option.



Add external testers for your app

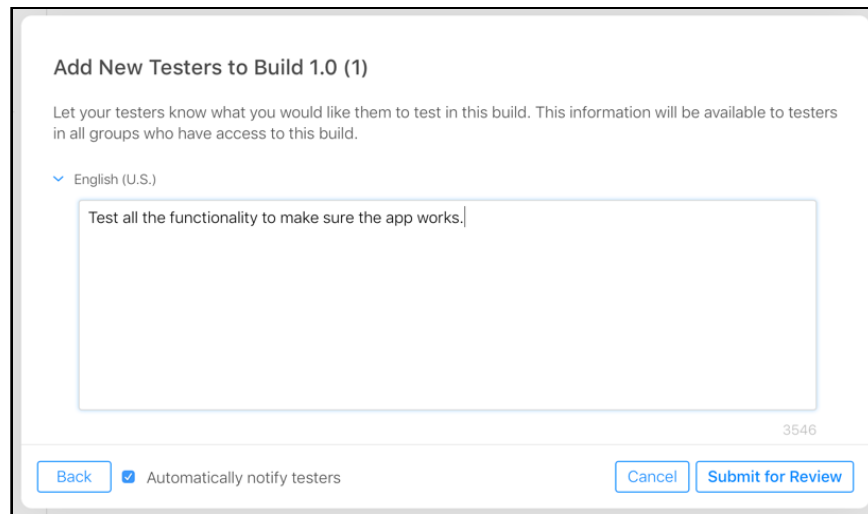
- This shows a new dialog where you can add external testers by e-mail address. Add one or more testers here.

The screenshot shows a dialog box titled 'Add New Testers to Build 1.0 (1)'. It says 'Add testers to test the build 1.0 (1) without adding them to a group.' Below is a table with columns 'Email', 'First Name', and 'Last Name'. The first row is filled with 'fahimf@...', 'Fahim', and 'Farook'. There are two empty rows below. At the bottom, there is a checkbox for 'Automatically notify testers' and 'Cancel' and 'Next' buttons.

The dialog for adding new testers

► When you are done adding testers, click the **Next** button.

This shows another screen which asks for sign in information in case your app requires a login. This is so that the Apple personnel reviewing your beta build have all the necessary information in order to test your app. Since *StoreSearch* does not require any login information, you can indicate that no sign in is required and proceed to the next screen.



The submit for beta review dialog

► Enter some text here explaining what you need tested (and perhaps what changes were made since the last build if this is a subsequent beta build) and then click **Submit for Review** to start the beta review process off.

Now, you wait :]

Generally, you will hear back from Apple within a day or two. If the Apple review team finds any issues with your app, they will let you know and you would need to fix the issues and submit another build and go through the review process again till you succeed.

If the app passes review, then it will be in "Testing" state for external testing and your external testers will be notified via e-mail that a new build is ready for them to test. This will start off the beta cycle for your app. Congratulations!

At this point, all you have to do is wait for feedback from your beta testers, fix any issues they find or make changes based on beta feedback, release another beta build and rinse and repeat till you are certain that your app is ready to be submitted to Apple for release.

Submit to the App Store

When your beta testing is complete, you can submit the final build which passed beta testing directly to Apple for App Store review. This way, you bypass the potential for accidental introduction of any new bugs when you create a new build.

You don't even have to upload a build or do anything. Since the build has already been uploaded to the Apple servers (when you uploaded it for beta testing), you simply have to move on to the next stage in the process using the correct build.

At this point, if you only entered the bare minimum information to add an app to iTunes Connect, you might need to provide some additional information for the app as well.

➤ Go to your app's detail screen and then select the **App Store** tab and then the **App Information** option from the left sidebar.

On this screen, make sure that the **Category** values are filled in correctly for your app. You can select any category from the dropdown here but do try to make sure that the categories you select are relevant for your app.

➤ Select the **Pricing and Availability** option from the left sidebar and set a price for your app.

You need to do the above two steps only once for every new app. All the other information changes you make below, might have to be done for every new release.

➤ Next, click on the **Prepare for Submission** link on the left sidebar with a yellow circle icon next to it. This is where you enter all the relevant information for each version that you submit to the App Store. Here, you'd need to complete at least the following information (you can fill in more values than the listed ones - but these are the mandatory ones):

- You can upload up to five screenshots and three 30-second movie per device. At a minimum, you need to supply screenshots for the 5.5-inch iPhones, and the 12.9-inch iPad Pro. If you do not provide screenshots or videos for the smaller screen devices, the assets from the larger screen devices will be scaled down for the smaller screens.
- 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 description that will be visible on the store

- The build to submit - this is the build that passed your beta testing. You can click on the plus icon (or use the link in the box) to get a list of uploaded builds and select the correct one from there.
- A 1024×1024 icon image
- Copyright information
- The version number
- The app rating - this is to identify whether your app contains potentially offensive content. You have to select from a list of items to determine your final content rating.
- Your contact details. Apple will contact you at this address if there are any problems with your submission.
- Sign-in information. If your app requires a user login to test its functionality, provide the necessary demo user name and password here. If a demo login is not required, remember to uncheck the **Sign-in required** checkbox. Otherwise, you will get an error when you try to submit the app.
- Notes for the reviewer. These are optional but a good idea if the reviewer needs to do anything special in order to test your app.
- When your app should become available

If your app supports multiple languages, then you can also supply a translated description, screenshots and even application name.

For more info and help, 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 App 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 probably think and then they're gone.

3. The screenshots. You need to have good screenshots that are exciting - 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 up to three 15 to 30-second videos that show 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.

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**.

If you missed any information, you will get an error message at the top of the screen indicating there are errors. The error message will be accompanied by a link which takes you to the page with the missing (or invalid) information. Also, the fields with missing information will be highlighted in red or have a red circle with an exclamation point next to the title (or both). This will help you identify what information needs to be filled in, or corrected.

Once you fix the issues, save and try submitting again. Sometimes, you have to do this multiple times before you finally succeed :]

Once you successfully submit your app, it enters 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 start at the bottom of the app review queue once again.

If after your app gets approved, you want to upload a new version of your app, the steps are largely the same. You change the version in Xcode (and change the build number), upload the new version to iTunes Connect, update the changed information in the Prepare for Submission screen and re-submit.

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.

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 iOS programming, and software development in general. I had a lot of fun writing these chapters and I hope you had a lot of fun reading them!

Because this book is packed with tips and information, you may want to go through it again in a few weeks, just to make sure you've picked up on everything!

The world of mobile app development 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 *StoreSearch*: 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 iOS: 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/. They are really worth it!
- The team at raywenderlich.com and I 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: store.raywenderlich.com

Stuck?

If you are 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. However, the 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, please 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