Fully updated for iOS 7!

iOS Apprentice

SECOND EDITION

Tutorial 4: StoreSearch

By Matthijs Hollemans

The iOS Apprentice:
StoreSearch

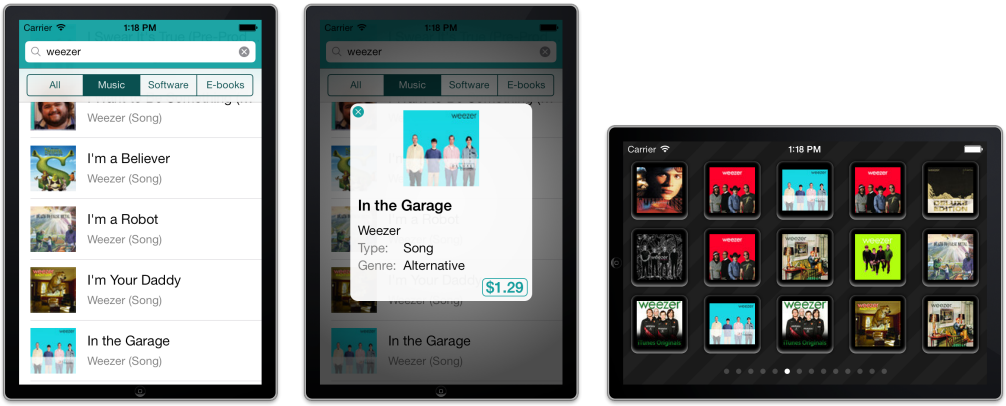
By Matthijs Hollemans

Table of Contents

One of the most common things that mobile apps do is talking to a server on the internet. If you're writing mobile apps, you need to know how to upload and download data. In this lesson you'll learn how to do HTTP GET requests to a web service, how to parse JSON data, and how to download files such as images.

You are going to build an app that lets you search the iTunes store. Of course, your iPhone already has apps for that ("App Store" and "iTunes Store" to name two), but what's the harm in writing another one? Apple has made a web service available for searching the entire iTunes store and you'll be using that to learn about networking.

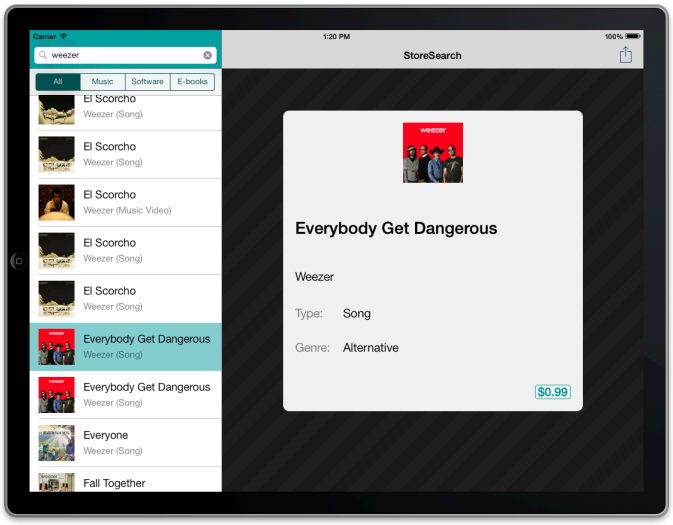
The finished app will look like this:



The finished StoreSearch app

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

There is also an iPad version of the app:



The app on the iPad

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

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

This chapter fills in the missing pieces and rounds off the knowledge you have obtained from the previous tutorials. You will also learn how to distribute your app to beta testers (so-called Ad Hoc Distribution) and how to submit it to the App Store.

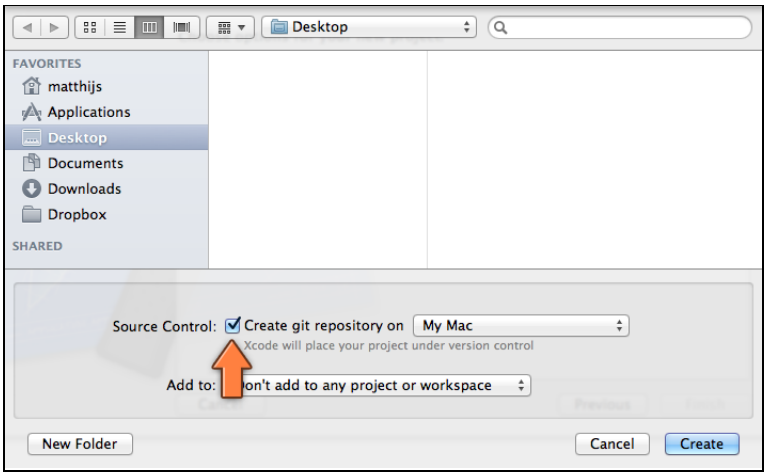
There's a lot of work ahead of you, so let's get started!

In the beginning...

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

- Product Name: **StoreSearch**
- Organization Name: your name
- Company Identifier: com.yourname
- Class Prefix: leave this empty
- Devices: iPhone

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



Creating a Git repository for the project

Git and version control

Git is a so-called **revision control system**. In short, Git allows you to make snapshots of your work so you can always go back later and see a history of what you did. Its principle is similar to Xcode's Snapshots feature but it offers a lot of extra goodies that are important when you're working on the same code with multiple people.

Imagine what happens if two programmers changed the same source file at the same time. Things will probably go horribly wrong! It's quite likely your changes will accidentally be overwritten by a colleague's. I once had a job where I had to shout down the hall to another programmer, "Are you using file X?" just so we wouldn't be destroying each other's work.

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

Git is not the only revision control system out there but it's the most popular one for iOS. A lot of iOS developers share their source code on GitHub (https://github.com), a free collaboration site that uses Git as its engine.

Another popular system is Subversion, often abbreviated as SVN. Xcode has built-in support for both Git and Subversion.

In this tutorial I'll show you some of the basics of using Git. Even if you work alone and you 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! – code.

In the previous tutorials you used storyboards for the app's user interface, but for StoreSearch you will use nibs. A **nib**, also called a xib, is very much like a storyboard except that it only contains the design for a single screen. That means you cannot make segues between different view controllers. Nibs, on the other hand, can also be used for other things than just view controllers, such as individual views or table view cells.

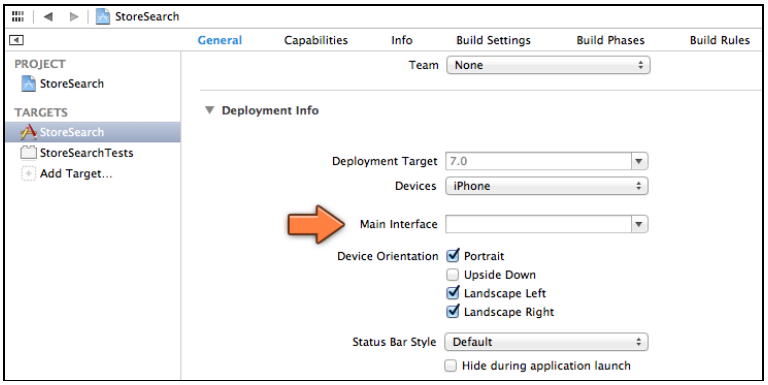
Storyboards would have worked just fine for this app, but I want to show you that nibs are also a valid choice. In practice, many apps consist of a combination of nibs and storyboard files, so it's good to know how to work with both.

Older versions of Xcode gave you the choice between making your new projects storyboard- or nib-based, but as of Xcode 5 all new projects use storyboards. To use nibs you have to perform a few additional steps:

- Remove the **Main.storyboard** file from the project. Choose **Move to Trash** because you won't be needing this anymore.

- Also remove the **ViewController.h** and **ViewController.m** files from the project.

- In the **Project Settings** screen in the **Deployment Info** section, remove the text from the **Main Interface** field so that it is empty:

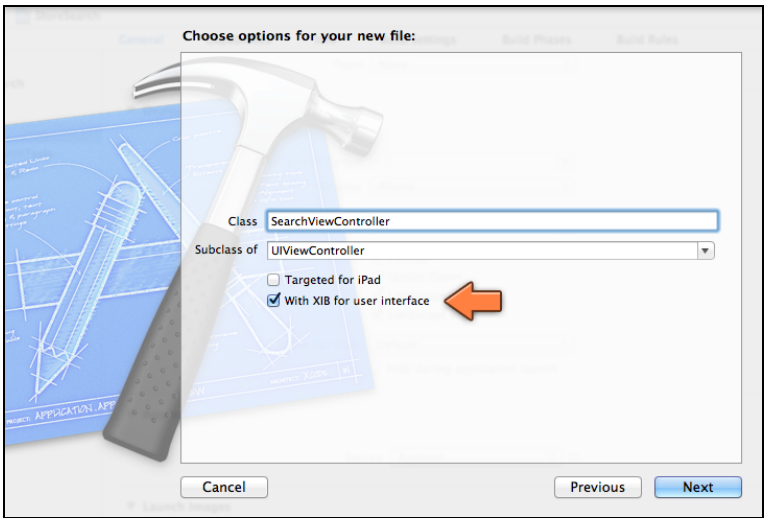


Removing the reference to the storyboard

- Run the app. You should get no error messages but the screen remains black. That's because there is nothing for the app to display.

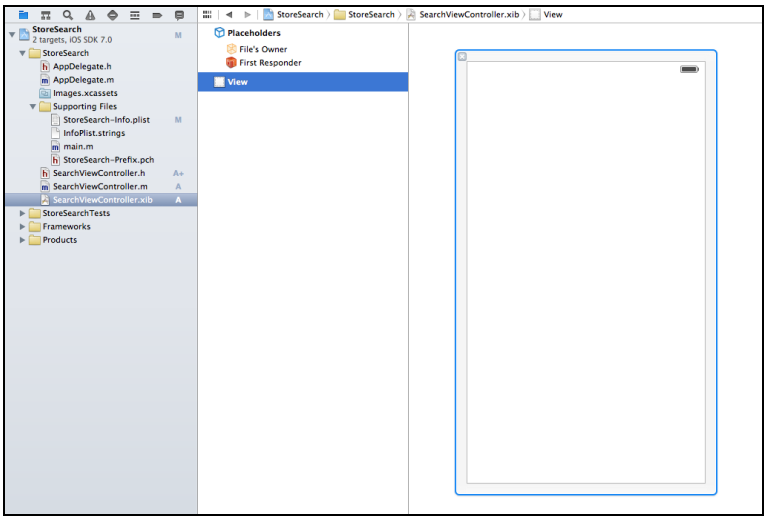
The first screen in StoreSearch will have a table view with a search bar, so let's make the view controller for that screen.

➤ Add a new file to the project named **SearchViewController**. Make it a subclass of a regular `UIViewController` (not table view controller!). Be sure to put a checkmark in front of the **With XIB for user interface** option.



Enabling the "With XIB for user interface" option

After you press Next, Xcode adds not two but three new files to the project: **SearchViewController.h**, **.m**, and **.xib**. The last one is the nib file with the user interface for this new view controller:



The SearchViewController.xib file

Adding the view controller and the nib file is by itself not enough. You also need to tell the app to load this view controller and display it on the screen. The best place for this is in the app delegate.

➤ Change **AppDelegate.h** to the following:

```
#import <UIKit/UIKit.h>

@class SearchViewController;

@interface AppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;
@property (strong, nonatomic) SearchViewController
                                          *searchViewController;

@end
```

➤ In **AppDelegate.m**, add an import at the top:

```
#import "SearchViewController.h"
```

➤ Change the `didFinishLaunchingWithOptions` method:

```
- (BOOL)application:(UIApplication *)application
     didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
  self.window = [[UIWindow alloc] initWithFrame:
                                  [[UIScreen mainScreen] bounds]];

  self.searchViewController = [[SearchViewController alloc]
            initWithNibName:@"SearchViewController" bundle:nil];

  self.window.rootViewController = self.searchViewController;
  [self.window makeKeyAndVisible];
  return YES;
}
```

This is boilerplate code that you'll find in just about any app that uses nibs. First it creates the `UIWindow` object. Every app needs at least one window and that window needs to have a `rootViewController`. For this app, that is the `SearchViewController`, which is created here by loading it from the nib with the same name. Finally, you make the window and everything in it visible.

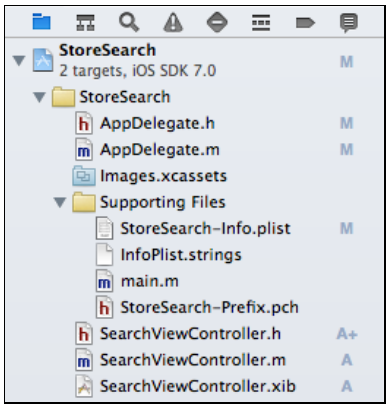
➤ For good measure, run the app to make sure everything works. You should see a white screen with the status bar on top.

The app shows the contents of the nib

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

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



Xcode shows the files that are modified

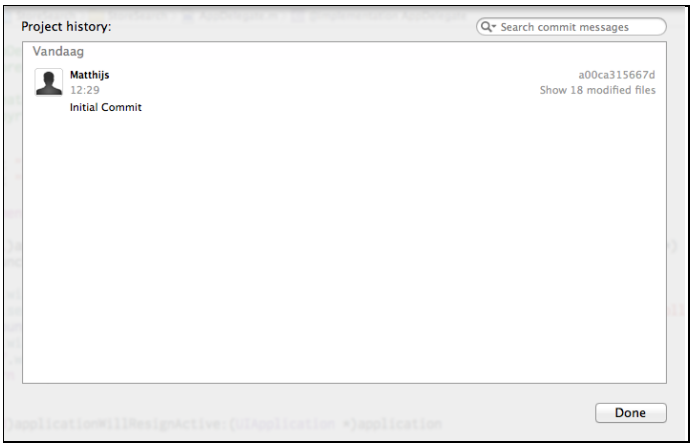
If you don't see these icons, then choose the **Source Control → Refresh Status** option from the Xcode menu bar. (If that gives an error message or still doesn't work, simply restart Xcode. That's a good tip in general: if Xcode is acting weird, restart it.)

An M means the file has been modified since the last "commit" and an A (or question mark) means that this is a file that has been added since then. So what is a commit?

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

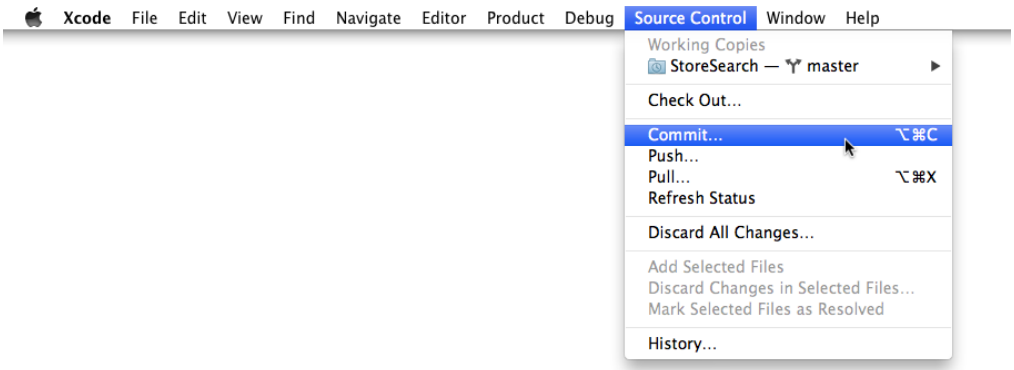
When you created the project, Xcode made the initial commit. You can see that in the Project History window.

› Choose **Source Control → History...**



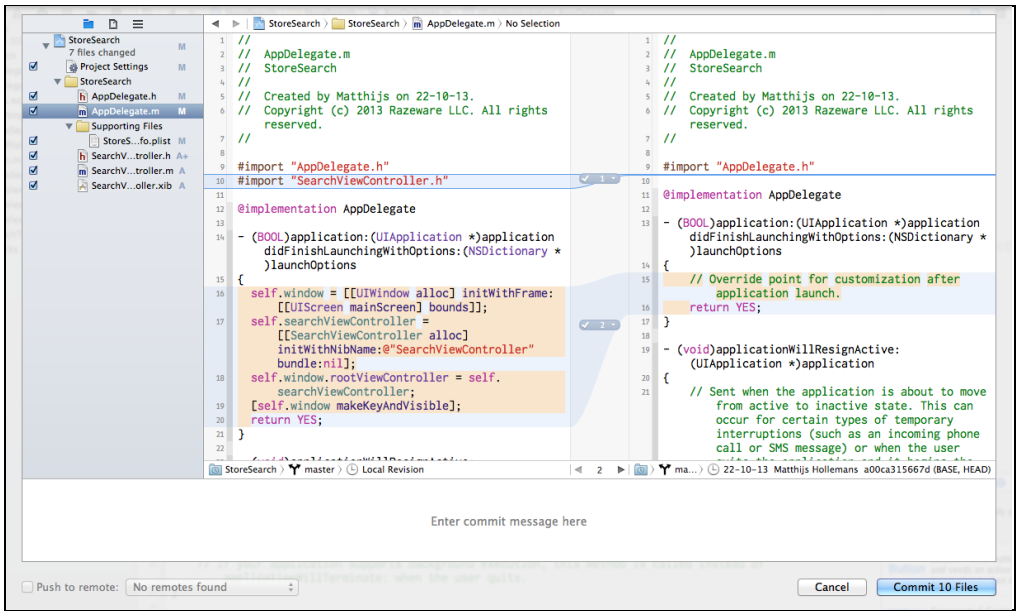
The history of commits for this project

› Let's commit the change you just made. Close the history window. From the **Source Control** menu, choose **Commit**...:



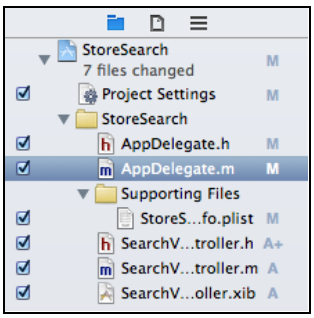
The Commit menu option

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



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

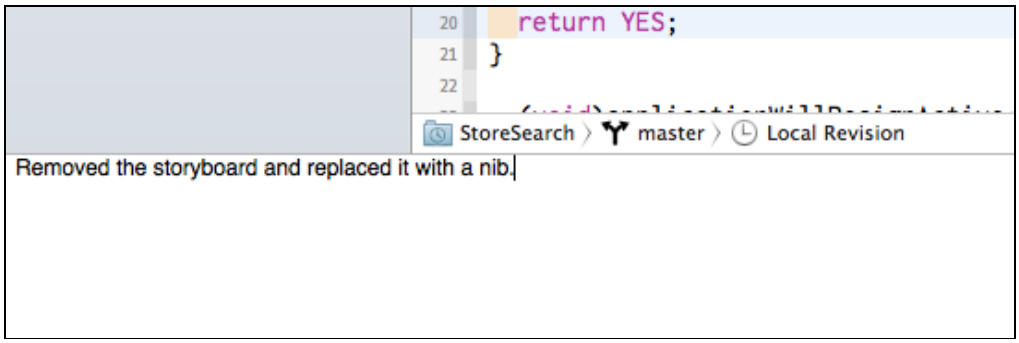
› Check the boxes for all the files on the left-hand side, because you want to include them all in the commit:



Selecting the files to 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: **Removed the storyboard and replaced it with a nib.**

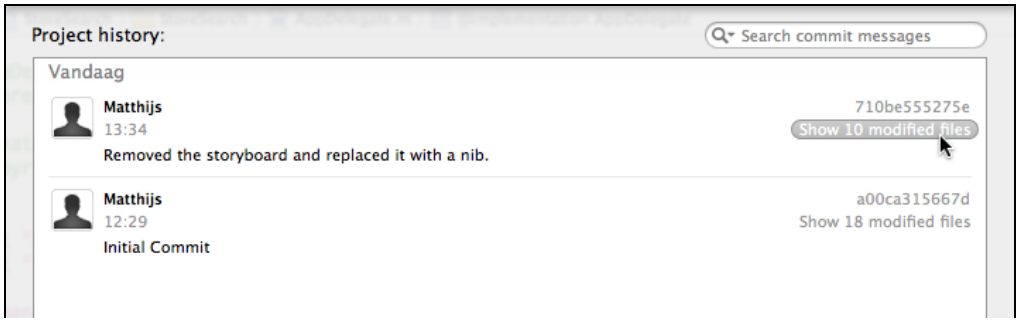


Writing the commit message

› Press the **Commit 10 Files** button. You'll see that in the project navigator the M and A icons are gone (at least until you make the next change).

By the way, even there are only seven files in the list, you did delete two source files and the storyboard, so in total 10 files were modified. The deleted files also count.

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

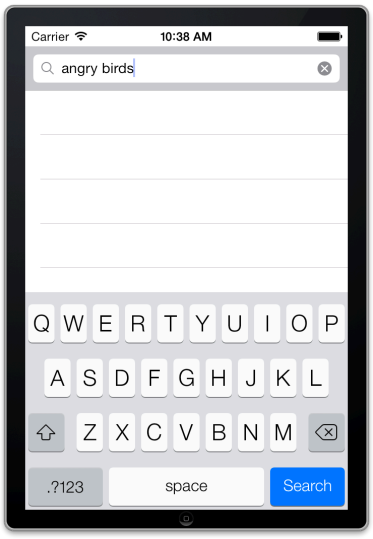


Your commit is listed in the project history

If you click on **Show 10 modified files**, Xcode will show you what has changed with that commit. You'll be doing commits on a regular basis and by the end of the tutorial you'll be a pro at it.

Creating the UI

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



The app with a search bar and table view

Even though this screen uses the familiar table view, it is not a *table* view controller but a regular `UIViewController`. You are not required to use a `UITableViewController` if you have a table view and for this app I will show you how to do without.

UITableViewController vs. UIViewController

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

For example, when a table cell contains a text field, tapping that text field will bring up the on-screen keyboard. `UITableViewController` automatically scrolls the cells out of the way of the keyboard so you can always see what you're typing. You don't get that behavior for free when you're using a plain `UIViewController`, so if you want this feature you'll have to program it yourself.

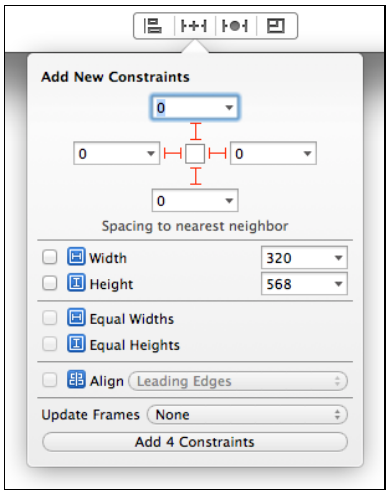
`UITableViewController` has a big restriction: its main view must be a `UITableView` that takes up the entire space (except for a possible navigation bar at the top, or a toolbar or tab bar at the bottom). If your screen consists of just a `UITableView`, then it makes sense to make it a `UITableViewController`, but if you want to have other views as well, a `UIViewController` is your only option.

That's the reason you're not using a `UITableViewController` in this app. You have another view besides the table view, a `UISearchBar`. It is possible to put

> the search bar *inside* the table view as a special header view, but for this app it will always be sitting on top.

› Open **SearchViewController.xib** and drag a new **Table View** into the view.

› Make the Table View as big as the main view (320 by 568 points) and then use the **Pin menu** at the bottom to attach the Table View to the edges of the screen:



Creating constraints to pin the Table View

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

› In the **Spacing to nearest neighbor** section, select the four red T-bars to make four constraints, one on each side of the Table View. Keep the spacing values at 0.

This will pin the Table View to the edges of its superview. Now the table will always fill up the entire screen, regardless of whether you're running the app on a 3.5-inch device or the larger 4-inch devices.

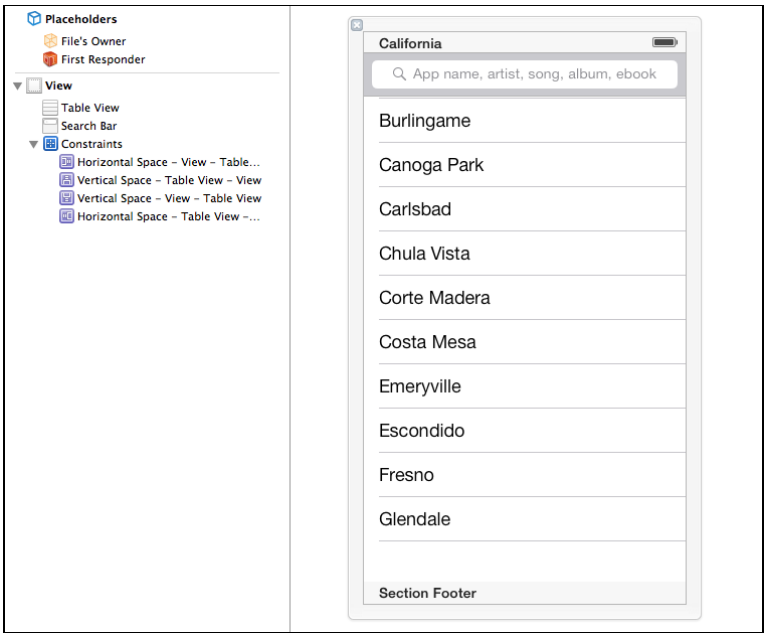
› Click the **Add 4 Constraints** button to finish.

› Drag a **Search Bar** component into the view. (Be careful to pick the Search Bar and not "Search Bar and Search Display Controller".) Place it at Y = 20 so it sits right under the status bar.

You don't need to add any constraints for the Search Bar. Remember that if you don't specify any constraints for a view, Auto Layout creates *automatic* constraints that give the view a fixed position and size. That's good enough for this Search Bar.

› In the **Attributes inspector** for the Search Bar, change the **Placeholder** text to **App name, artist, song, album, ebook**.

The view controller's design should look like this:



The search view controller with Search Bar and Table View

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

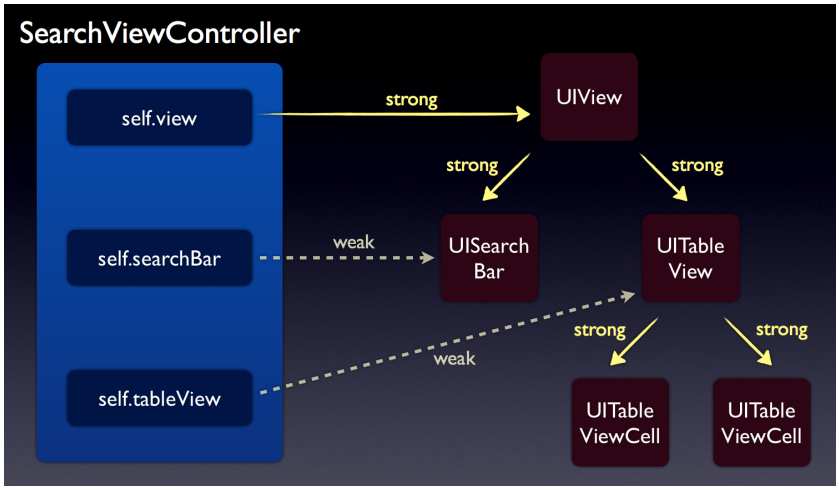
▸ Add the following to **SearchViewController.m**. It goes into the class extension at the top of the file:

```
@interface SearchViewController ()
@property (nonatomic, weak) IBOutlet UISearchBar *searchBar;
@property (nonatomic, weak) IBOutlet UITableView *tableView;
@end
```

Recall that as soon as an object no longer has any strong pointers, it goes away (it is deallocated) and all the weak pointers become `nil`. Per Apple's recommendation you've been making your outlets weak pointers. You may be wondering, if the pointers to these view objects are weak, then won't the objects get deallocated too soon?

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

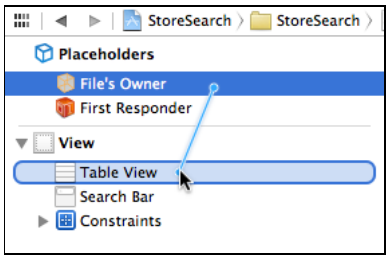
Answer: Views are always part of a view hierarchy and they will always have an owner with a strong pointer: their superview. In this screen, the `SearchViewController`'s main view object will hold a reference to both the search bar and the table view. This is done inside UIKit and you don't have to worry 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 pointers

➤ Open the nib and connect the Search Bar and the Table View to their respective outlets.

The quickest way to do this is to **Ctrl-drag** from **File's Owner** to the object that you want to connect:



Connecting the table view to the outlet

File's Owner

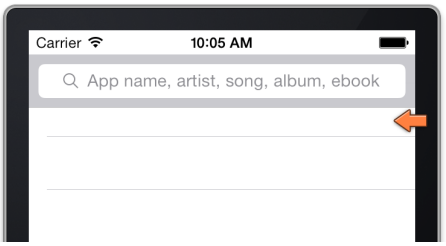
A nib, the user interface of a view controller, is *owned* by that view controller. This means the view controller will load that nib file, put all of its objects on the screen, and make the connections. The view controller will dispose of the nib file when the screen closes. File's Owner, therefore, refers to the view controller object.

File's Owner provides a convenient way to connect objects from the nib to outlets on the view controller. You simply Ctrl-drag from File's Owner to an object to hook up the outlet. Dragging the other way around, from a UI element to File's Owner, makes it possible to connect action methods and delegates.

If you're ever wondering who the File's Owner is for a nib, then click File's Owner to select it and open the Identity inspector (the third button at the top

of the inspector area). The Custom Class field will show you the owner's name. If you do that for this nib, you'll see that it says **SearchViewController**.

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.



That's not so strange because in the nib you put the Search Bar on top of the table. To fix this you could nudge the Table View down a few pixels. However, according to the design guidelines for iOS 7 the content of a view controller should take up the entire screen space. So it's better to leave the size of the Table View alone and to make the Search Bar partially translucent to let the contents of the table cells shine through. But it would still be nice to see the first few rows in their entirety.

You can compensate for this with the **content inset** attributes of the Table View. Unfortunately, with Auto Layout enabled this attribute is unavailable in Interface Builder (at least in Xcode 5.0) so you'll have to do this from code.

➤ Add the following line to `viewDidLoad` in **SearchViewController.m**:

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  self.tableView.contentInset = UIEdgeInsetsMake(64, 0, 0, 0);
}
```

This tells the table view to add a 64-point margin at the top, made up of 20 points for the status bar and 44 points for the Search Bar. Now the first row will always be visible, and when you scroll the table view the cells still go under the search bar. Nice.

Doing fake searches

Before you can search the iTunes store, it's good to understand how the `UISearchBar` component works. In this section you'll get the text to search for from the search bar and use that to put some fake search results into the table view. Once you've got that working, you can build in the web service. Small steps!

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

Listening to the search bar is done – how else? – with a delegate.

➤ Add the following to the `@interface` line in **SearchViewController.m**:

```
@interface SearchViewController () <UITableViewDataSource,
                        UITableViewDelegate, UISearchBarDelegate>
```

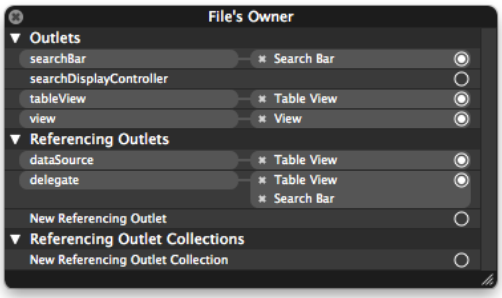
Not only did you add the `UISearchBarDelegate`, you also added the `UITableViewDataSource` and `UITableViewDelegate` protocols. That wasn't necessary in the previous tutorials because you used a `UITableViewController` there, which by design already conforms to these protocols. For this app you're using a regular view controller and therefore you'll have to hook up the data source and delegate protocols yourself.

➤ Open the nib and **Ctrl-drag** from the Search Bar to File's Owner. Connect to **delegate**.

➤ **Ctrl-drag** from the Table View to File's Owner. Connect to **dataSource**. Repeat and connect to **delegate**.

Note that you connected something to File's Owner's "delegate" twice. The way Interface Builder presents this is a little misleading: the `delegate` outlet is not from File's Owner (i.e. `SearchViewController`), but belongs to the thing that you Ctrl-dragged from.

So you connected the `SearchViewController` to the `delegate` outlet on the Search Bar and also to the `delegate` (and `dataSource`) outlets on the Table View:



The connections from File's Owner to the other objects

➤ Run the app. Whoops! It immediately crashes. Xcode's Debug pane says:

```
StoreSearch[24699:f803] -[SearchViewController tableView:
numberOfRowsInSection:]: unrecognized selector sent to instance
0x6e4f510
```

You told the table view that the `SearchViewController` is its data source but you didn't actually implement any of those data source methods yet.

That's what this error means: `SearchViewController` is missing the method named `tableView:numberOfRowsInSection:`.

Whenever you get the "unrecognized selector" error, you forgot to implement a method somewhere (or you may have misspelled it). Fortunately, the error message also tells you which object is missing the method, and what the method name is supposed to be.

➤ Add the following to the bottom of **SearchViewController.m**:

```
#pragma mark - UITableViewDataSource

- (NSInteger)tableView:(UITableView *)tableView
             numberOfRowsInSection:(NSInteger)section
{
  return 0;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  return nil;
}
```

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 the table view crashing.

Often you can declare to conform to a protocol without implementing any of its methods. This works fine for `UISearchBarDelegate`, for example, but obviously not in the case of `UITableViewDataSource`! A protocol can have `@optional` and `@required` methods and if you forget a required method, a crash is your reward. Fortunately, Xcode already warns about any missing delegate methods when you build the app. As always, pay attention to the warnings from Xcode!

➤ Run the app. It should no longer crash.

The `UISearchBarDelegate` protocol has a method `searchBarSearchButtonClicked:` that is invoked when the user taps the Search button on the keyboard. You will implement this method to put some fake data into the table. In a little while you'll make this method send a network request to the iTunes store to find songs, movies and ebooks that match the search text that the user typed, but let's not do too many new things at once!

➤ Add the following method to **SearchViewController.m**, below the table view stuff:

```
#pragma mark - UISearchBarDelegate
```

```
- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
  NSLog(@"The search text is: '%@'", searchBar.text);
}
```

▸ 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.

> **Tip:** I always put strings in between single quotes when I use `NSLog()` to print them. That way you can easily see whether there are any trailing or leading spaces in the string.

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

▸ Add an instance variable for the array:

```
@implementation SearchViewController
{
  NSMutableArray *_searchResults;
}
```

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

▸ Replace the `searchBarSearchButtonClicked:` method with:

```
#pragma mark - UISearchBarDelegate

- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
  _searchResults = [NSMutableArray arrayWithCapacity:10];

  for (int i = 0; i < 3; i++) {
    [_searchResults addObject:[NSString stringWithFormat:
                @"Fake Result %d for '%@'", i, searchBar.text]];
  }

  [self.tableView reloadData];
}
```

Here you allocate a new `NSMutableArray` object and put it into the `_searchResults` instance variable. This is done each time the user performs a search. If there was already a previous array with search results then it is thrown away and deallocated.

You add a string with some text into the array. Just for the fun of it, that is repeated 3 times so your data model will have three rows in it. The last statement in the method reloads the table view to make the new rows visible, which means you have to adapt the table view data source methods to read from this array as well.

➤ Change the `tableView:numberOfRowsInSection:` method to:

```
- (NSInteger)tableView:(UITableView *)tableView
             numberOfRowsInSection:(NSInteger)section
{
  if (_searchResults == nil) {
    return 0;
  } else {
    return [_searchResults count];
  }
}
```

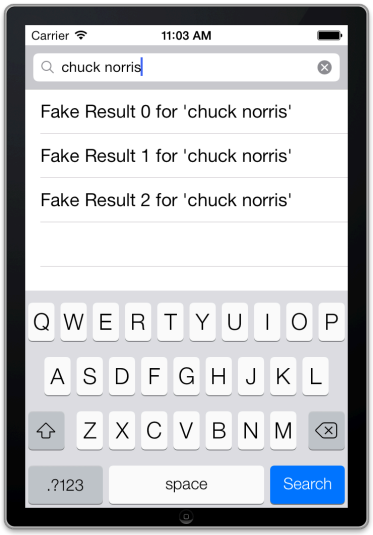
When the app first starts up, the value of the `_searchResults` variable will be `nil` because no search is done yet and you never allocated the array. In that case, you return 0 for the number of rows. Just remember that this "search results equals nil" is a special case that means the user did not search for anything yet. That's different from the case where the user did search and no matches were found.

➤ Finally, change the `tableView:cellForRowAtIndexPath:` method to:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  static NSString *CellIdentifier = @"SearchResultCell";

  UITableViewCell *cell = [tableView
              dequeueReusableCellWithIdentifier:CellIdentifier];
  if (cell == nil) {
    cell = [[UITableViewCell alloc]
                  initWithStyle:UITableViewCellStyleDefault
                  reuseIdentifier:CellIdentifier];
  }

  cell.textLabel.text = _searchResults[indexPath.row];
  return cell;
}
```

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

➤ Run the app. If you search for anything, a couple of fake results get added to the data model and are shown in the table. Search for something else and the table view updates with new fake results.



The app shows fake results when you search

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

➤ Add the following line at the start of `searchBarSearchButtonClicked:`

```
- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
  [searchBar resignFirstResponder];
  . . .
```

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

As of iOS 7 you can also configure the table view to dismiss the keyboard with a gesture.

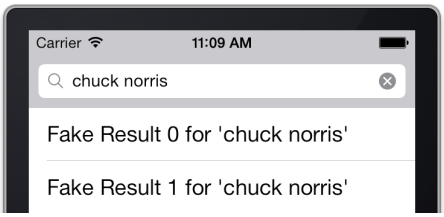
➤ In the nib, select the Table View. Go to the **Attributes inspector** and set **Keyboard** to **Dismiss interactively**.

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

➤ Add the following method to the bottom of the file. This is part of the `SearchBarDelegate` protocol.

```
- (UIBarPosition)positionForBar:(id <UIBarPositioning>)bar
{
  return UIBarPositionTopAttached;
}
```

Now the app looks a lot smarter:



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

Improving the data model

So far you've added `NSString` objects to the `_searchResults` array, but that's a bit limited. The search results that you'll get back from the iTunes store include the product name, the name of the artist, a link to an image, the purchase price, and much more. You can't fit all of that in a single string, so let's create a new class to hold this data.

➤ Add a new file to the project. Choose the **Objective-C class** template. Name the new class **SearchResult** and make it a subclass of **NSObject**.

➤ Add two properties to the new `SearchResult` class. This time you will add them to the regular `@interface` section (not in a class extension) because other objects should be able to use these properties.

```
@interface SearchResult : NSObject

@property (nonatomic, copy) NSString *name;
@property (nonatomic, copy) NSString *artistName;

@end
```

The properties are marked `copy` rather than `strong`. When you assign a value to such a property, it will first make a copy of the object and then treat that as a strong reference. It is a good idea to use `copy` for classes such as `NSString` and `NSArray` that have a mutable subclass (`NSMutableString`, `NSMutableArray`). For more info, see the section "Copy and assign properties" in the MyLocations tutorial.

For now you'll stick to just these two properties but in a little while you'll add several others.

In the `SearchViewController` you will no longer add `NSStrings` to the `_searchResults` array, but an instance of `SearchResult`.

➤ First, add an import to the top of **SearchViewController.m**:

```
#import "SearchResult.h"
```

➤ Change the search bar delegate method to:

```
- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
  [searchBar resignFirstResponder];

  _searchResults = [NSMutableArray arrayWithCapacity:10];

  for (int i = 0; i < 3; i++) {
    SearchResult *searchResult = [[SearchResult alloc] init];
    searchResult.name = [NSString stringWithFormat:
                                    @"Fake Result %d for", i];
    searchResult.artistName = searchBar.text;
    [_searchResults addObject:searchResult];
  }

  [self.tableView reloadData];
}
```

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

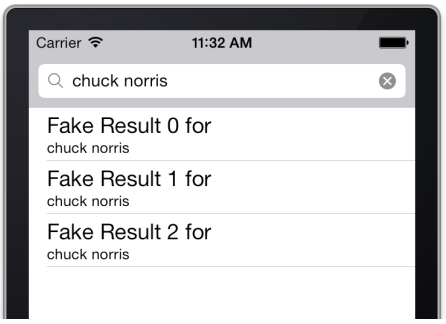
➤ At this point, the `cellForRowAtIndexPath` method still expects the array to contain strings so the app will crash if you don't also update that:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  static NSString *CellIdentifier = @"SearchResultCell";

  UITableViewCell *cell = [tableView
              dequeueReusableCellWithIdentifier:CellIdentifier];
  if (cell == nil) {
    cell = [[UITableViewCell alloc]
                      initWithStyle:UITableViewCellStyleSubtitle
                      reuseIdentifier:CellIdentifier];
  }

  SearchResult *searchResult = _searchResults[indexPath.row];
  cell.textLabel.text = searchResult.name;
  cell.detailTextLabel.text = searchResult.artistName;
```

```
    return cell;
}
```

Instead of a regular table view cell this is now using a "subtitle" cell style and you put the contents of the `artistName` property into the detail (subtitle) text label.

▸ When you run the app, it looks like this:



Fake results in a subtitle cell

Nothing found

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

1. The user did not perform a search yet. This is the case when the `_searchResults` instance variable is `nil`.
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 that there were no results. If you display nothing at all the user may wonder whether the search was actually performed or not.

Even though the app doesn't do any actual searching yet – everything is fake – there is no reason why you cannot fake the latter situation as well. For the sake of good taste, the app will return 0 results when the user searches for "justin bieber", just so you know the app can handle this kind of situation.

▸ Change `searchBarSearchButtonClicked:` to the following:

```
- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
  [searchBar resignFirstResponder];

  _searchResults = [NSMutableArray arrayWithCapacity:10];

  if (![searchBar.text isEqualToString:@"justin bieber"]) {
    for (int i = 0; i < 3; i++) {
```

```
      SearchResult *searchResult = [[SearchResult alloc] init];
      searchResult.name = [NSString stringWithFormat:
                                    @"Fake Result %d for", i];
      searchResult.artistName = searchBar.text;
      [_searchResults addObject:searchResult];
    }
  }

  [self.tableView reloadData];
}
```

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

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

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

➤ Change `cellForRowAtIndexPath` to:

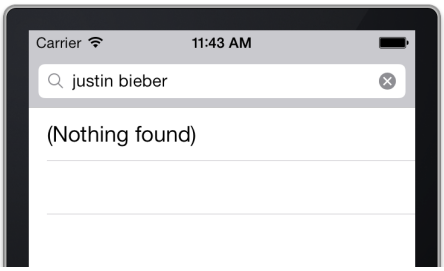
```
- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  static NSString *CellIdentifier = @"SearchResultCell";

  UITableViewCell *cell = [tableView
            dequeueReusableCellWithIdentifier:CellIdentifier];
  if (cell == nil) {
    cell = [[UITableViewCell alloc]
                    initWithStyle:UITableViewCellStyleSubtitle
                    reuseIdentifier:CellIdentifier];
  }

  if ([_searchResults count] == 0) {
    cell.textLabel.text = @"(Nothing found)";
    cell.detailTextLabel.text = @"";
  } else {
    SearchResult *searchResult = _searchResults[indexPath.row];
    cell.textLabel.text = searchResult.name;
    cell.detailTextLabel.text = searchResult.artistName;
  }
  return cell;
}
```

That alone is not enough. The call to `[_searchResults count]` now returns 0 because there is nothing in the array. That means the data source's `numberOfRowsInSection` will also return 0 and the table view stays empty.

▸ Change `numberOfRowsInSection` to:

```
- (NSInteger)tableView:(UITableView *)tableView
             numberOfRowsInSection:(NSInteger)section
{
  if (_searchResults == nil) {
    return 0;
  } else if ([_searchResults count] == 0) {
    return 1;
  } else {
    return [_searchResults count];
  }
}
```

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



One can hope...

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

▸ To fix that, add the following methods below the other table view stuff:

```
#pragma mark - UITableViewDelegate

- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  [tableView deselectRowAtIndexPath:indexPath animated:YES];
}

- (NSIndexPath *)tableView:(UITableView *)tableView
             willSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
```

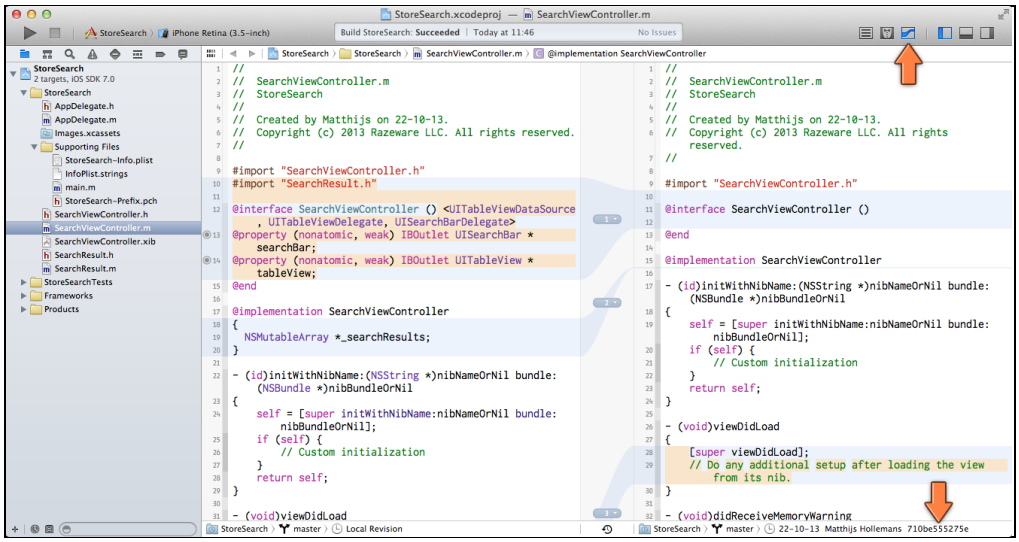
```
    if ([_searchResults count] == 0) {
      return nil;
    } else {
      return indexPath;
    }
  }
```

The `didSelectRowAtIndexPath` method will simply deselect the row with an animation, while `willSelectRowAtIndexPath` makes sure that you can only select rows with actual search results.

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

▸ This is a good time to commit the app. Go to **Source Control** → **Commit** (or press the **Cmd+Option+C** keyboard shortcut). Make sure all the files are selected, review your changes, type a good commit message – something like "Added a search bar and table view. The search puts fake results in the table for now." – and press the **Commit** button.

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



Viewing revisions in the Version editor

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

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

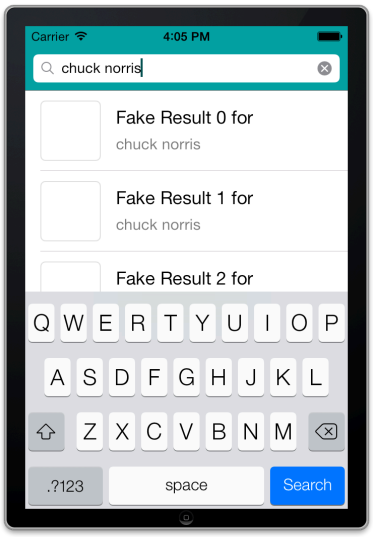
You can find the project files for the first part of this app under **01 - Search Bar** in the tutorial's Source Code folder.

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

Custom table cells in Interface Builder

In the previous tutorials you used prototype cells to create your own table view cell layouts. Unfortunately, prototype cells only work with storyboards and not with nibs. Not all is lost, though. You can create a new nib file with the design for the cell and load your table view cells from that. The principle is very similar to prototype cells.

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



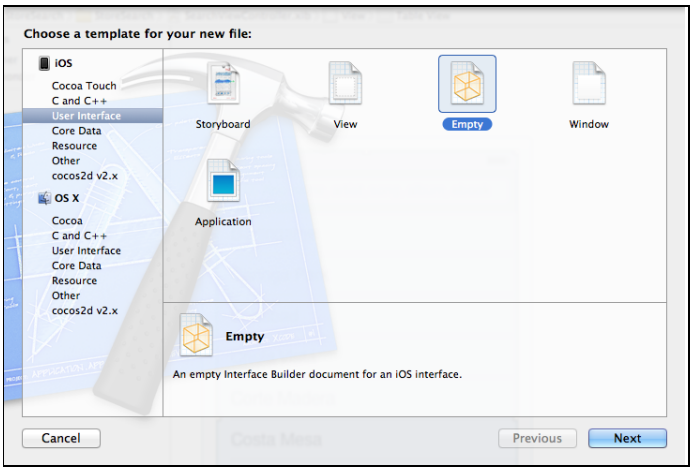
The app with better looks

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

So far you've made a nib for an entire view controller but you can also make a nib that contains just a `UITableViewCell`. A nib is really nothing more than a collection of frozen views.

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

› Add a new file to the project. Choose the **Empty** template from the **User Interface** category. This will create a new nib without anything in it.



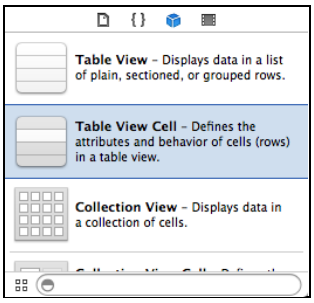
Adding an empty nib to the project

› In the next step, choose **Device Family: iPhone**. (There really isn't any fundamental difference between a nib for iPhone or iPad, except that if you were to choose iPad here, the nib would be bigger.)

› Finally, save the new file as **SearchResultCell.xib**. (Remember that the file extension for a nib file is **.xib**, not .nib.)

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

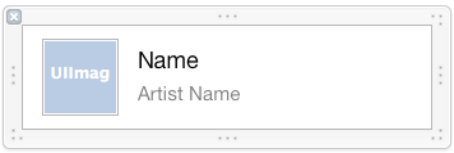
› From the Object Library, drag a new **Table View Cell** into the canvas:



The Table View Cell in the Object Library

› Select the new Table View Cell and go to the **Size inspector**. Type 80 in the **Height** field.

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



The design of the cell

› The Image View is positioned at X:15, Y:10, Width:60, Height:60.

› The **Name** label is at X:90, Y:13, Width:230, Height:27. Its font is **System 18.0**.

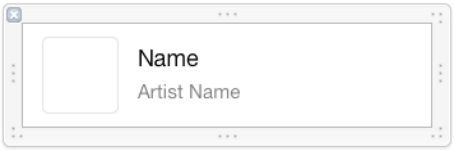
› The **Artist Name** label is at X:90, Y:42, Width:230, Height:21. Font is **System 15.0** and Color is black with 50% opacity.

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

The image view will hold the artwork for the found item, such as an album cover, book cover, or an app icon. It may take a few seconds for these images to be loaded, so until then it's a good idea to show a placeholder image. You've already added a bunch of image files to the project and one of them is that placeholder image.

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

The completed cell design now looks like this:



The completed cell design

Now that you have a nib for this cell, you have to tell the app to use it.

› In **SearchViewController.m**, add these lines to the bottom of `viewDidLoad`:

```
UINib *cellNib = [UINib nibWithNibName:@"SearchResultCell"
                                bundle:nil];

[self.tableView registerNib: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 `dequeueReusableCellWithIdentifier:` 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 `cellForRowAtIndexPath` to:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:@"SearchResultCell"];

  if ([_searchResults count] == 0) {
    cell.textLabel.text = @"(Nothing found)";
    cell.detailTextLabel.text = @"";
  } else {
    SearchResult *searchResult = _searchResults[indexPath.row];
    cell.textLabel.text = searchResult.name;
    cell.detailTextLabel.text = searchResult.artistName;
  }

  return cell;
}
```

You were able to replace this chunk of code,

```
static NSString *CellIdentifier = @"SearchResultCell";

UITableViewCell *cell = [tableView
              dequeueReusableCellWithIdentifier:CellIdentifier];
if (cell == nil) {
    cell = [[UITableViewCell alloc]
                    initWithStyle:UITableViewCellStyleSubtitle
                    reuseIdentifier:CellIdentifier];
}
```

with just one statement:

```
UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:@"SearchResultCell"];
```

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

➤ Run the app and do a (fake) search. Hmm, that doesn't look too good:

Uh oh...

There are two problems here. First of all, these rows aren't 80 points high. The table view isn't smart enough to figure out that these custom cells need to be higher. Fortunately this is easily fixed.

➤ Add the following line to `viewDidLoad`:

```
self.tableView.rowHeight = 80;
```

The second problem is more serious: the text doesn't end up in the right place and overlaps the image view.

Exercise. Any ideas why? □

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

Every table view cell – even custom ones 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", "Value1", "Value2"). If you use them on custom cells then these labels get in the way of your own labels.

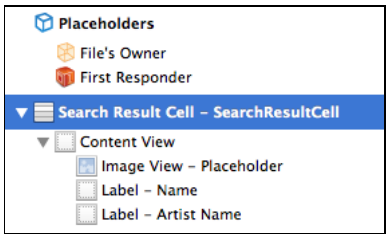
So here you shouldn't use `textLabel` and `detailTextLabel` to put text into the cell, but make your own properties for your own labels.

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

➤ Add a new file to the project, **Objective-C class** template. Name it **SearchResultCell** and make it a subclass of **UITableViewCell**. This creates the .h and .m files to accompany the .xib 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:



The cell's top-level view is now a SearchResultCell

From now on, whenever you do `dequeueReusableCellWithIdentifier:`, the table view will return an object of type `SearchResultCell`.

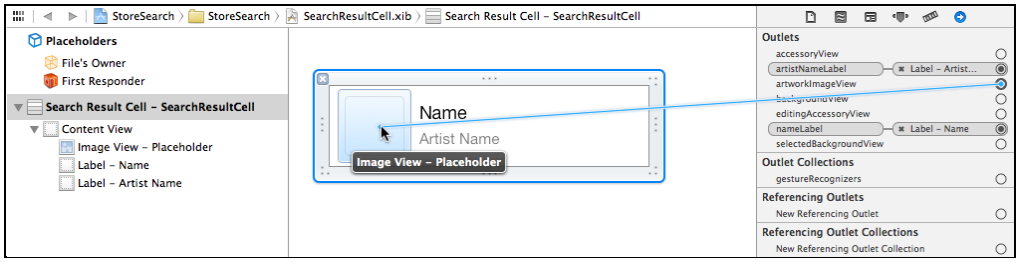
› Add the following properties to **SearchResultCell.h**:

```
@property (nonatomic, weak) IBOutlet UILabel *nameLabel;
@property (nonatomic, weak) IBOutlet UILabel *artistNameLabel;
@property (nonatomic, weak) IBOutlet UIImageView
                                        *artworkImageView;
```

Again you make these outlets `weak`, but this time you add them to the .h file and not the .m file. That's because now you want another object – the `SearchViewController` – to access these properties and therefore they must be publicly visible.

› Hook these outlets up to the respective labels and image view in the nib.

You might be tempted to Ctrl-drag from File's Owner to these objects but that won't work. In this particular case there is no File's Owner. Instead, you should connect the labels and image view to the Search Result Cell object (visible in the sidebar).



Connect the labels and image view to Search Result Cell, not File's Owner

Tip: For some reason, Xcode gave me grief when I tried Ctrl-dragging from Search Result Cell to the labels. If this happens to you then use the Connections inspector or the Assistant editor to hook up the outlets.

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

› Add an import at the top of **SearchViewController.m**:

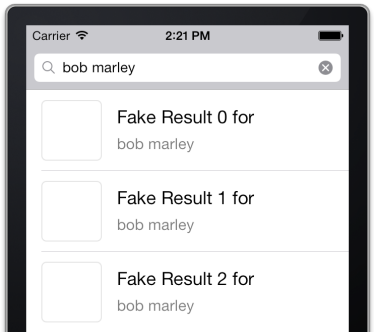
```
#import "SearchResultCell.h"
```

➤ Change `cellForRowAtIndexPath` to:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  SearchResultCell *cell = (SearchResultCell *)[tableView
        dequeueReusableCellWithIdentifier:@"SearchResultCell"];

  if ([_searchResults count] == 0) {
    cell.nameLabel.text = @"(Nothing found)";
    cell.artistNameLabel.text = @"";
  } else {
    SearchResult *searchResult = _searchResults[indexPath.row];
    cell.nameLabel.text = searchResult.name;
    cell.artistNameLabel.text = searchResult.artistName;
  }

  return cell;
}
```

The biggest change is the first line. Previously this returned a `UITableViewCell` object but now that you've changed the class in the nib, you're guaranteed to always receive a `SearchResultCell`. Given that cell, you can put the name and artist name from the search result into the proper labels.

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



Much better!

Note: You could also have written the line that dequeues the cell as follows:

```
SearchResultCell *cell = (SearchResultCell *)[tableView
        dequeueReusableCellWithIdentifier:@"SearchResultCell"
        forIndexPath:indexPath];
```

The new bit is a second parameter that passes along the index-path. This variant of the dequeue method was added in iOS 6 and it lets the table view

be a bit smarter, but it only works when you have registered a nib with the table view. That's exactly what you did in `viewDidLoad`, so it's safe to use this new version of `dequeueReusableCellWithIdentifier` here.

There are a few more things to improve. Notice that you've been using the string literal `@"SearchResultCell"` in a few different places? It's generally better to create a *constant* for such occasions. Suppose you – or one of your co-workers – renamed the reuse identifier in one place (for whatever reason), then you also have to remember to change it in all the other places where `@"SearchResultCell"` is used. It's better to limit those changes to one single spot by using a symbolic name instead.

➤ Add the following at the top of the file, below the imports:

```
static NSString * const SearchResultCellIdentifier =
                                              @"SearchResultCell";
```

This defines a symbolic name, `SearchResultCellIdentifier`, with the value `@"SearchResultCell"`. Should you want to change this value, then you only have to do it here and any code that uses `SearchResultCellIdentifier` 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 word `SearchResultCellIdentifier`.

➤ Anywhere in **SearchViewController.m**, replace the string `@"SearchResultCell"` with `SearchResultCellIdentifier`.

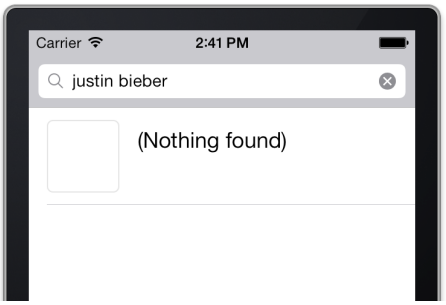
For example, `viewDidLoad` will now look like this:

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  self.tableView.contentInset = UIEdgeInsetsMake(64, 0, 0, 0);
  self.tableView.rowHeight = 80;

  UINib *cellNib = [UINib
          nibWithNibName:SearchResultCellIdentifier bundle:nil];
  [self.tableView registerNib:cellNib
       forCellReuseIdentifier:SearchResultCellIdentifier];
}
```

➤ Run the app to make sure everything still works.

A new "Nothing Found" cell

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



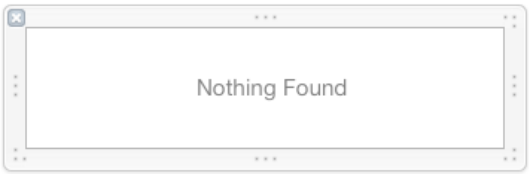
The Nothing Found label now draws like this

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

➤ Add another nib file to the project. Again this will be an **Empty** nib. Name it **NothingFoundCell.xib**.

➤ Drag a new **Table View Cell** into the canvas. Set its **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**. Center the label in the cell. It should look like this:



Design of the Nothing Found cell

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

➤ Add to the top of **SearchViewController.m**:

```
static NSString * const NothingFoundCellIdentifier =
                                              @"NothingFoundCell";
```

➤ Add to `viewDidLoad`:

```
cellNib = [UINib nibWithNibName:NothingFoundCellIdentifier
                         bundle:nil];
```

```
[self.tableView registerNib:cellNib
     forCellReuseIdentifier:NothingFoundCellIdentifier];
```

➤ And finally, change `cellForRowAtIndexPath` to:

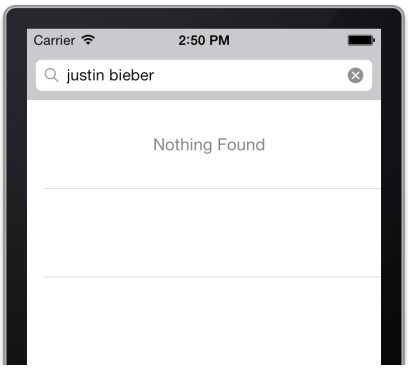
```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  if ([_searchResults count] == 0) {
    return [tableView dequeueReusableCellWithIdentifier:
              NothingFoundCellIdentifier forIndexPath:indexPath];
  } else {
    SearchResultCell *cell = (SearchResultCell *)[tableView
      dequeueReusableCellWithIdentifier:
              SearchResultCellIdentifier forIndexPath:indexPath];

    SearchResult *searchResult = _searchResults[indexPath.row];
    cell.nameLabel.text = searchResult.name;
    cell.artistNameLabel.text = searchResult.artistName;

    return cell;
  }
}
```

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

Remember that in `willSelectRowAtIndexPath` 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.

➤ Open **NothingFoundCell.xib** and 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.

The search results for Justin Bieber now look like this:



The new Nothing Found cell in action

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 "Now uses custom cells for search results."

Changing the look of the app

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

▸ Add the following method at the top of **AppDelegate.m**:

```
- (void)customizeAppearance
{
  UIColor *barTintColor = [UIColor colorWithRed:20/255.0f
                    green:160/255.0f blue:160/255.0f alpha:1.0f];
  [[UISearchBar appearance] setBarTintColor:barTintColor];

  self.window.tintColor = [UIColor colorWithRed:10/255.0f
                      green:80/255.0f blue:80/255.0f alpha:1.0f];
}
```

Note that this changes the appearance of *all* `UISearchBars` in the application. You only have one, but if you have several then this changes the whole lot in one swoop.

The `[UIColor colorWithRed: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` method, however, accepts values between 0.0 and 1.0, so you have to divide these numbers by 255.0 to scale them down to that range.

Be careful when dividing integers

Dividing by 255 (without the .0 or the "f") is not a good idea. Doing the following will result in a color that is completely black:

```
UIColor *color = [UIColor colorWithRed:20/255 green:160/255
                         blue:160/255 alpha:1.0f];
```

The value 255 is an integer and dividing the integer 20 by the integer 255 will always result in a value that is 0 because integers cannot have numbers behind the decimal point.

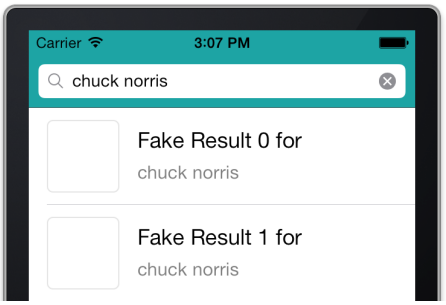
To solve this problem, you divide by 255.0f, which is a floating-point number and they can have decimals. Now you're no longer dealing with just integers and the result will also be a floating-point value.

This is just one of those Objective-C programming gotchas that you have to be aware of.

▸ Call this new method from `application:didFinishLaunchingWithOptions:`, directly after the line that creates the window (this is important!).

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
  self.window = [[UIWindow alloc] initWithFrame:
                                [[UIScreen mainScreen] bounds]];
  [self customizeAppearance];
  . . .
```

▸ 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 work as 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:

```
AppDelegate *appDelegate = (AppDelegate *)
                    [UIApplication sharedApplication].delegate;
```

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

In my opinion, it's better to design your code the other way around: the app delegate may do a certain amount of initialization, but then it gives any data model objects to the root view controller, and hands over control. The root view controller passes these data model objects to any other controller that needs them, and so on. This is also called *dependency injection*. I described this principle in the section "Passing around the context" in the MyLocations tutorial.

Currently, tapping a row gives it a gray selection. This doesn't go so well with the teal-colored theme so I'd like the row selection to have the same bluish-green tint. That's very easy to do because all table view cells have a `selectedBackgroundView` property. The view from that property is placed on top of the cell's background, but below the other content, when the cell is selected.

➤ Add the following method to the bottom of **SearchResultCell.m**:

```
- (void)awakeFromNib
{
  [super awakeFromNib];

  UIView *selectedView = [[UIView alloc]
                                      initWithFrame:CGRectZero];
  selectedView.backgroundColor = [UIColor colorWithRed:20/255.0f
                    green:160/255.0f blue:160/255.0f alpha:0.5f];
  self.selectedBackgroundView = selectedView;
```

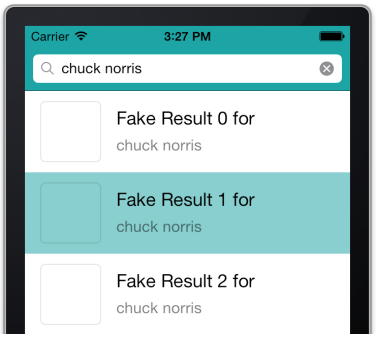
```
}
```

The `awakeFromNib` method is called immediately after this cell object has been loaded from the nib. You can use it to do additional initialization. Why don't you do that in an init method, such as `initWithCoder:`? No particular reason, I just wanted to show you that `awakeFromNib` existed (it's also less typing!).

It's worth noting that `awakeFromNib` is called some time after `initWithCoder:` and after all of the other objects from the nib have been created. For example, in `initWithCoder:` the `self.nameLabel` outlet will still be `nil` but in `awakeFromNib` it will be properly hooked up to the corresponding `UILabel` object.

Don't forget to first call `[super awakeFromNib]`, which is required. If you forget, then the superclass `UITableViewCell` (or any of the other superclasses) may not get a chance to initialize themselves. It's always a good idea to call `[super methodName]` in a method that you're overriding (such as `viewDidLoad`, `viewWillAppear:`, and so on), unless the documentation says otherwise.

When you run the app, it should look as follows:



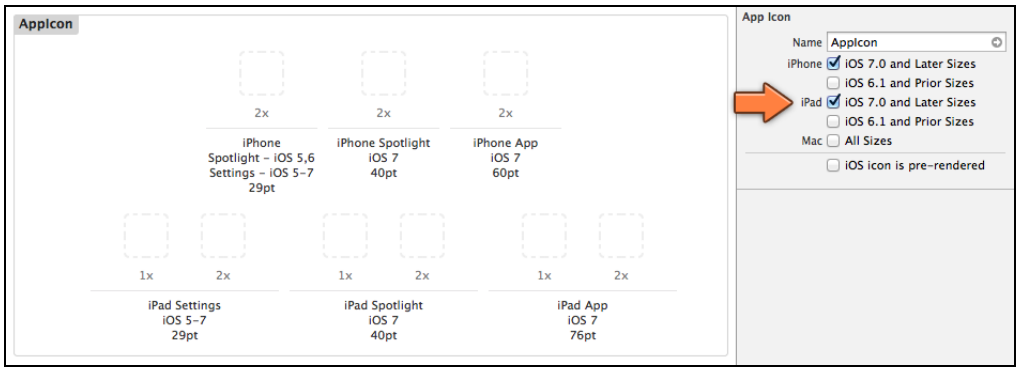
The selection color is now green

While you're at it, you might as well give the app an icon. (You cannot add the launch images yet because you'll make a small change to the design of the search screen later.)

➤ Open the asset catalog (**Images.xcassets**) and select the **AppIcon** group.

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

➤ Open the **Attributes inspector** and put a checkmark in front of **iPad, iOS 7.0 and Later Sizes**:

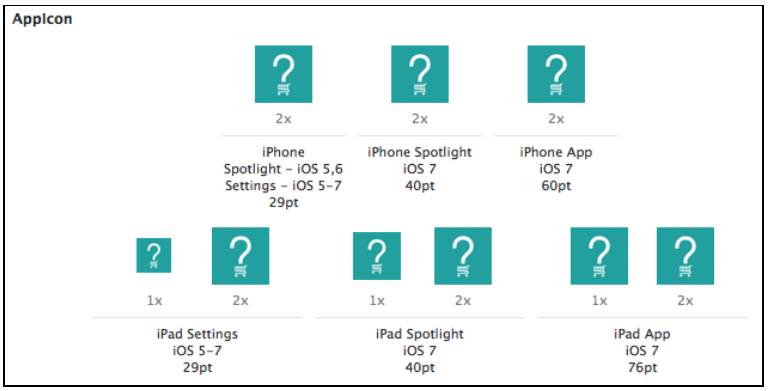


Enabling iPad icons

This adds six new slots for the iPad icons.

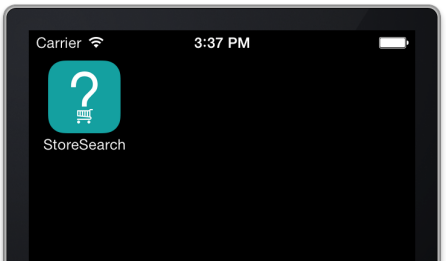
› Drag the images from the **Icon** folder from this chapter's resources into the slots.

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



All the icons in the asset catalog

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



The app icon

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

› Add the following line to `viewDidLoad` in **SearchViewController.m**:

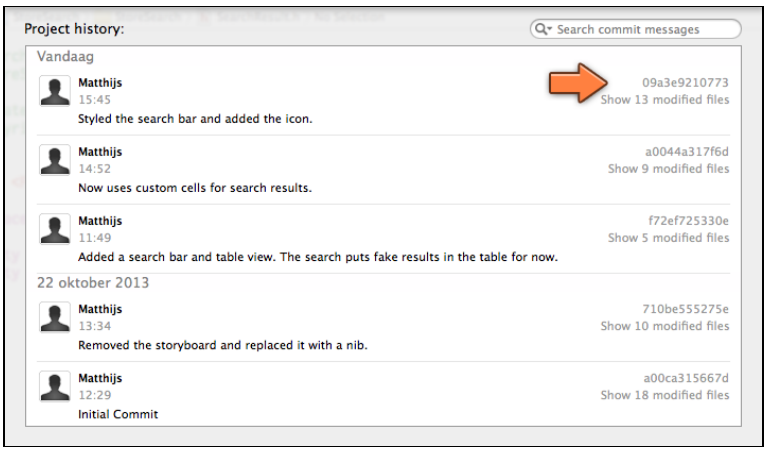
```
[self.searchBar becomeFirstResponder];
```

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

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

Tagging the commits

If you look through the various commits you've made so far, you'll notice a bunch of strange numbers, such as "09a3e9210773":



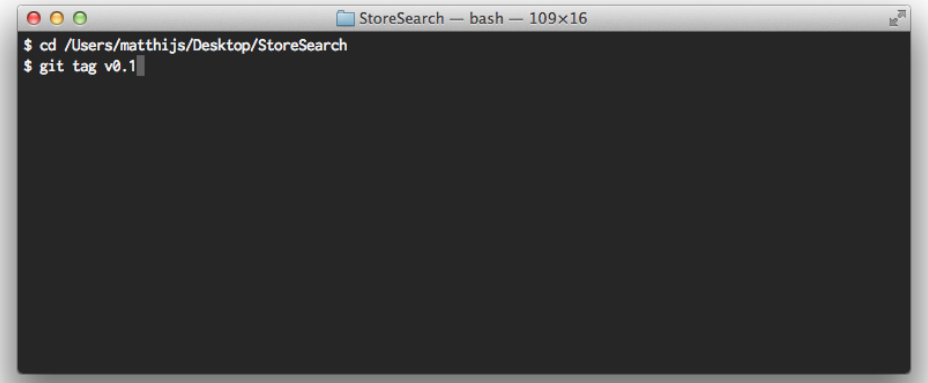
The commits are listed in the history window but have weird numbers

Those are internal numbers that Git uses to uniquely identify commits (known as the "hash"). However, such numbers aren't very nice for us humans so Git also allows you to "tag" a certain commit with a more friendly label. Unfortunately, at the time of writing, Xcode does not support this tag command. You can do it from a Terminal window, though.

› Open the **Terminal** (from **Applications/Utilities**).

› Type "**cd** " and from Finder drag the folder that contains the StoreSearch project into the Terminal. Then press Enter. This will make the Terminal go to your project directory.

› Type the command "**git tag v0.1**".

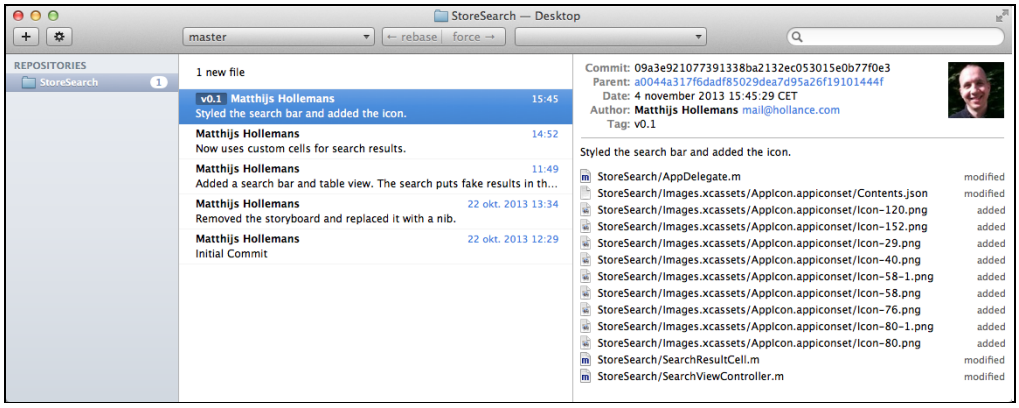


Doing git tag from the Terminal

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

> **Note:** If typing the git command in Terminal gives you a "command not found" error, then open the Xcode Preferences and under the **Downloads** tab install the **Command Line Tools**.

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



Viewing the Git repository with Gitbox

Xcode works quite well with Git but it only supports the basic features. To take full advantage of Git you'll probably need to learn how to use the Terminal or get a tool such as Gitbox or SourceTree (both on the Mac App Store), GitX (http://gitx.frim.nl/), or GitHub for Mac (http://mac.github.com/).

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

The debugger

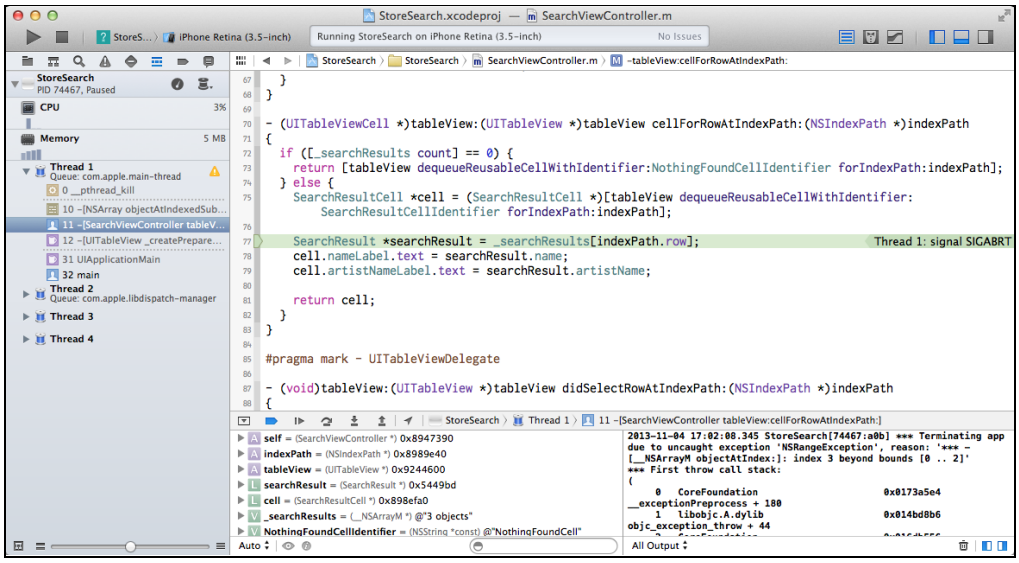
Xcode has a built-in **debugger**. Unfortunately, a debugger doesn't actually get the bugs out of your programs; it just lets them crash in slow motion so you can get a better idea of what is wrong. Like a police detective, the debugger lets you dig through the evidence after the damage has been done, in order to find the scoundrel who did it.

Let's introduce a bug into the app so that it crashes. Knowing what to do when your app crashes is very important. Thanks to the debugger, you don't have to stumble in the dark with no idea what just happened. Instead, you can use it to quickly pinpoint what went wrong and where. Once you know that, figuring out *why* it went wrong becomes a lot easier.

➤ Change **SearchViewController.m**'s `numberOfRowsInSection` method to:

```
- (NSInteger)tableView:(UITableView *)tableView
             numberOfRowsInSection:(NSInteger)section
{
  if (_searchResults == nil) {
    return 0;
  } else if ([_searchResults count] == 0) {
    return 1;
  } else {
    return [_searchResults count] + 1;
  }
}
```

➤ Now run the app and search for something. The app crashes and the Xcode window should change to something like this:



The Xcode Debugger appears when the app crashes

The crash is: **Thread 1: signal SIGABRT**. There are different types of crashes (you'll look at EXC_BAD_ACCESS later) but SIGABRT is actually a pretty good one to have. It means your app died in a controlled fashion. You did something you were not supposed to and UIKit or another framework caught this and politely terminated the app with an error message.

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

```
*** Terminating app due to uncaught exception 'NSRangeException',
reason: '*** -[__NSArrayM objectAtIndex:]: index 3 beyond bounds [0 ..
2]'
*** First throw call stack:
(
  0   CoreFoundation    0x0173a5e4 __exceptionPreprocess + 180
  1   libobjc.A.dylib   0x014bd8b6 objc_exception_throw + 44
  2   CoreFoundation    0x016db556 -[__NSArrayM objectAtIndex:]
   . . .
)
libc++abi.dylib: terminating with uncaught exception of type NSException
```

The first part of this message is very important: it tells you that the app was terminated because of an `NSRangeException`. In some programming languages **exceptions** are a commonly used error handling mechanism, but in Objective-C an exception is only thrown when some fatal error happens.

The bit that should pique your interest is this:

```
-[__NSArrayM objectAtIndex:]: index 3 beyond bounds [0 .. 2]
```

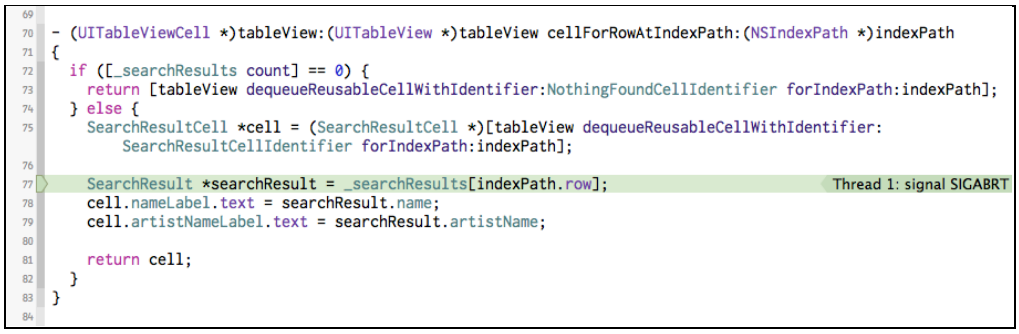
Apparently the error happened when the app sent the `objectAtIndex:` message to an `NSArray` object. The real object name here is `__NSArrayM`, which happens to be a special object that `NSArray` uses internally to do its heavy lifting. But even if you don't know that, you can still recognize this has something to do with an `NSArray`-type object.

> **Note:** The `objectAtIndex:` message is used to retrieve an object from an array. You usually write the following,
>
> ```
> object = array[index]
> ```
>
> but this is equivalent to doing:
>
> ```
> object = [array objectAtIndex:index]
> ```
>
> The `array[index]` notation is a recent addition to the language, so you'll still see a lot of code use the `objectAtIndex:` method. In practice it doesn't matter which one you use but most programmers find `array[index]` more natural because the `[]` notation is used for dealing with arrays in many programming languages.

According to the error message, the thing that made the app crash is that the index that was used on the array is 3, but the valid range is 0 through 2. In other words, the index is "out of bounds". That is a common error with arrays and you're likely to make this mistake a couple more times in your programming career.

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

Thankfully, you have the debugger to help you out. In the source code editor it already points at the offending line:



The debugger points at the line that crashed

Note that 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.

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

➤ Restore `numberOfRowsInSection` to what it was and then change the following line:

```
static NSString * const SearchResultCellIdentifier =
                                              @"NothingFoundCell";
```

This one is very sneaky. You've changed the reuse identifier, so dequeuing a cell will now return the contents of **NothingFoundCell.xib** where instead you expect to get a `SearchResultCell`. The compiler won't notice this mismatch because it happens during runtime.

➤ Run the app and search for something. The app crashes again with a SIGABRT and the error message is:

```
-[UITableViewCell nameLabel]: unrecognized selector sent to instance
0x8981c40
*** Terminating app due to uncaught exception
'NSInvalidArgumentException', reason: '-[UITableViewCell nameLabel]:
unrecognized selector sent to instance 0x8981c40'
. . .
```

There it is again, "unrecognized selector sent to instance". The code is assuming it is getting a `SearchResultCell` object, which has a `nameLabel` property. However, because you specified the wrong re-use identifier the app is actually getting a regular `UITableViewCell`, which does not have that `nameLabel` property. (NothingFoundCell.xib contains a regular `UITableViewCell` because you did not make a subclass for "NothingFoundCell".)

So when the code tries to do the following,

```
cell.nameLabel.text = searchResult.name;
```

the app doesn't know what you mean. The `cell` variable points at a `UITableViewCell` object and the app tries to send it a "nameLabel" message to access the property. There is no method to handle that message and the app bails out.

➤ Put the reuse identifier back to `@"SearchResultCell"` and a new outlet property to **SearchViewController.m**:

```
@property (nonatomic, weak) IBOutlet UISearchBar *searchBar2;
```

➤ Open **SearchViewController.xib** and **Ctrl-drag** from File's Owner to the Search Bar. Select **searchBar2** from the popup. Now the search bar is also connected to this new `searchBar2` outlet. (It's perfectly fine for an object to be connected to more than one outlet at a time.)

➤ Remove the `searchBar2` outlet property from the **.m** file.

This is a dirty trick on my part to make the app crash. The nib file 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. Again the dreaded "Thread 1: signal SIGABRT".

The Debug pane says:

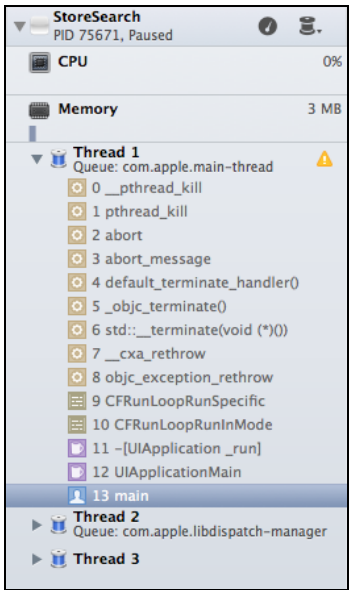
```
*** Terminating app due to uncaught exception 'NSUnknownKeyException',
reason: '[<SearchViewController 0xa27c600> setValue:forUndefinedKey:]:
this class is not key value coding-compliant for the key searchBar2.'
```

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 main.m?

That also isn't very useful. Xcode says the app crashed in **main.m**, 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.

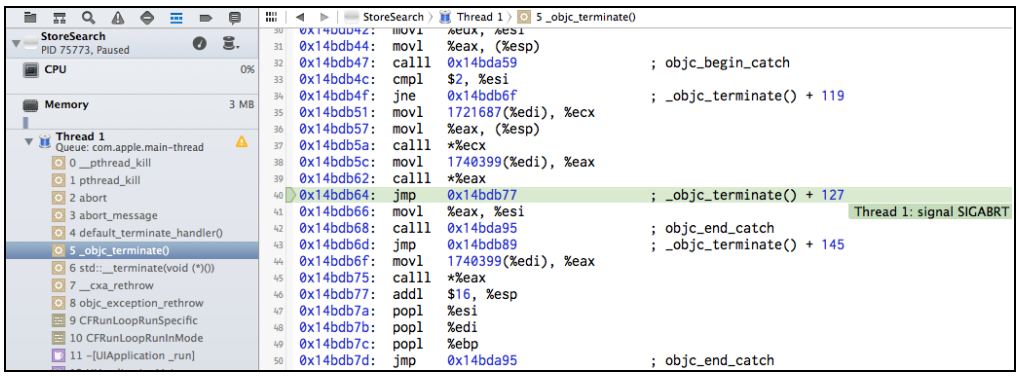
➤ Drag the slider at the bottom all the way to the right to see more info:



A more detailed call stack

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

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



You cannot look inside the source code of system libraries

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

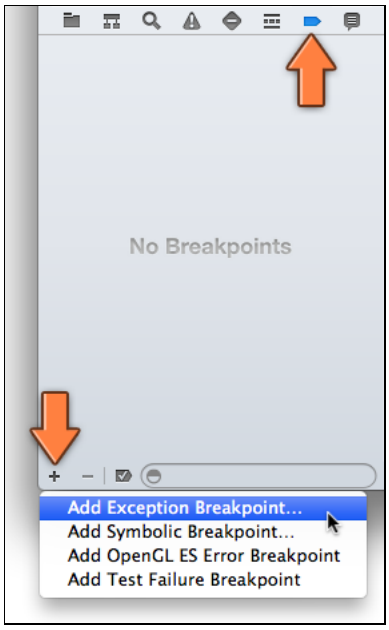
> A **breakpoint** is a special marker in your code that will pause the app and jump into the debugger. When your app hits a breakpoint, the app will pause at that exact spot. Then you can use the debugger to step line-by-line through

your code in order to run it in slow motion. That can be a handy tool if you really cannot figure out why something crashes.

You're not going to step through code in this tutorial, but you can read more about it in the Xcode Overview guide, in the section Debug Your App. You can find it in the iOS Developer Library. https://developer.apple.com/library/ios/

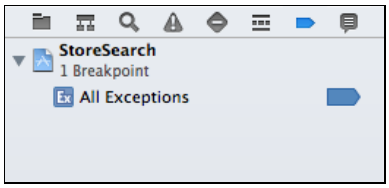
You are going to set a special breakpoint that is triggered whenever a fatal exception occurs. This will halt the program just as it is about to crash, which should give you more insight into what is going on.

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



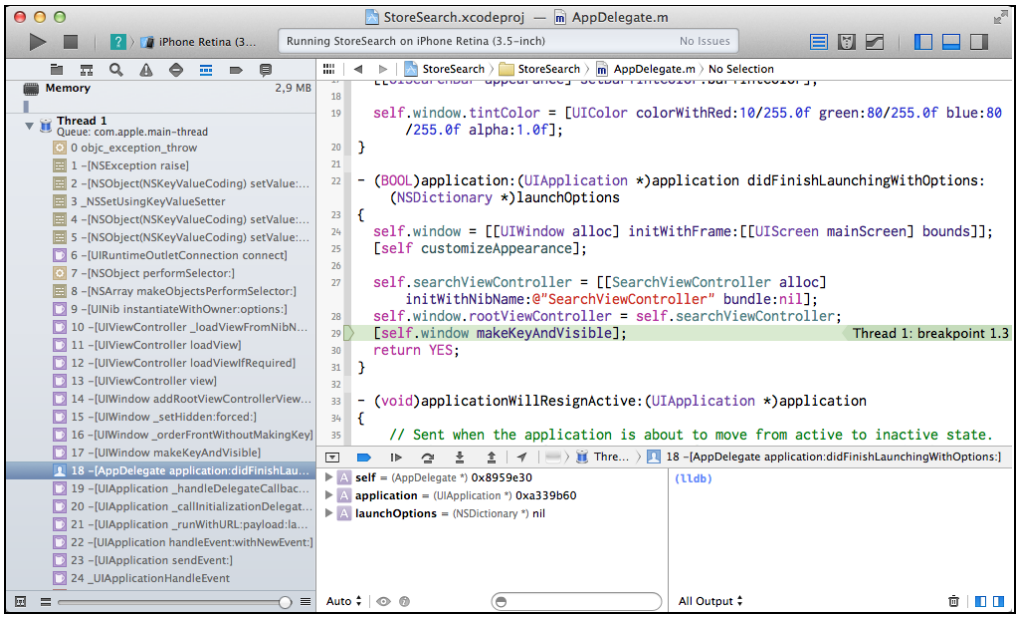
Adding an Exception Breakpoint

This will put a new breakpoint in the list:



After adding the Exception Breakpoint

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



Xcode now halts the app as the exception is being thrown

There are much more methods in the call stack now. The only one that has source code is the `application:didFinishLaunchingWithOptions:` method in your `AppDelegate` class. But you can see in the call stack that it doesn't end here.

Higher up, there is a call to something called `[UIViewController _loadViewFromNibNamed:bundle:]`, which gives a clue that this has something to do with loading a nib file. There is also a call to `[UIRuntimeOutletConnection connect]`. This is not some class that you've seen before but its name hints at the fact that it has something to do with outlet connections.

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 it's clear that the app crashes when it's loading the nib because the `searchBar2` outlet property no longer exists on the view controller.

› Open **SearchViewController.xib** and disconnect File's Owner from **searchBar2** to fix the crash.

Enabling the Exception Breakpoint means that you no longer get a useful error message in the Debug pane if you app crashes (because the exception was never actually thrown). If sometime later during development your app crashes on another bug, you may want to disable this breakpoint again to actually see the error message. You can do that from the Breakpoint navigator.

To summarize:

- If your app crashes with a 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 **main.m**, enable the Exception Breakpoint to get more info.
- If the app crashes with a SIGABRT but there is no error message, then disable the Exception Breakpoint and make the app crash again. (Alternatively, click the **Continue program execution** button from the debugger toolbar a few times. That will also show the error message.)

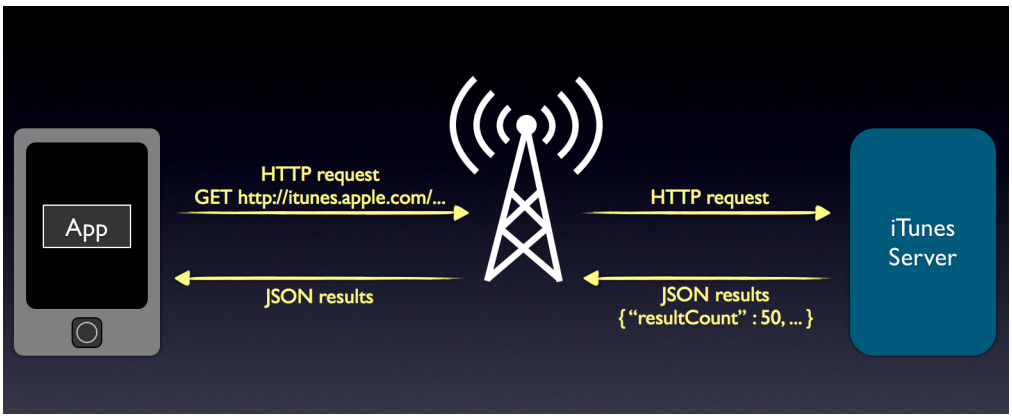
This should help you get to the bottom of most of your crashes!

It's all about the networking

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

The iTunes store sells a lot of products: songs, e-books, movies, software, TV episodes... you name it. You can sign up as an affiliate and earn a commission on each sale that happens because you recommended a product (even your own apps!). To make it easier for affiliates to find products, Apple made available a web service that queries the iTunes store. You're not going to sign up as an affiliate for this tutorial but you will use that free web service to perform searches.

So what is a **web service**? Your app (also known as the "client") will send a message over the network to the iTunes store (the "server") using the HTTP protocol. Because the iPhone can be connected to different types of networks – Wi-Fi or a cellular network such as 3G, EDGE or GPRS – the app has to "speak" a variety of networking protocols to communicate with other computers on the Internet. Fortunately you don't have to worry about any of that as the iPhone firmware will take care of this complicated subject matter. All you need to know is that you're using HTTP.



The HTTP requests fly over the network

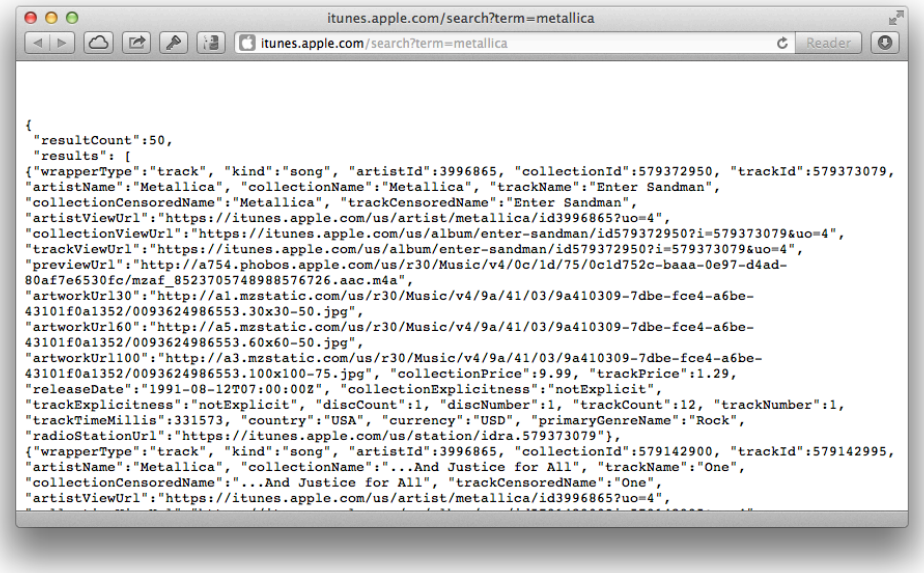
HTTP is the exact same protocol that your web browser uses when you visit a web site. In fact, you can play with the iTunes web service using a web browser. That's a great way to figure out how this web service works. This trick won't work with all

web services (some require "POST" requests instead of "GET" requests) but often you can get quite far with just a web browser.

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

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

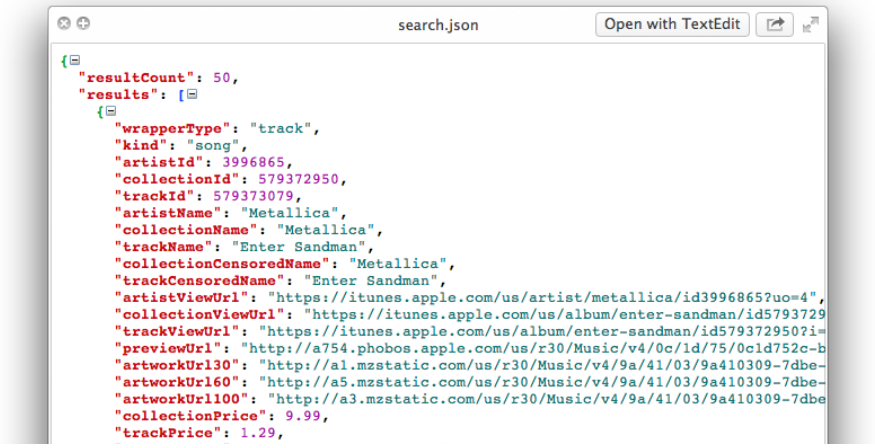
The browser should show something like this:



Using the iTunes web service from the Safari web browser

Those are the search results that the iTunes web service gives you. The data is in a format named JSON, which stands for JavaScript Object Notation. JSON is commonly used to send structured data back-and-forth between servers and clients (i.e. apps). Another data format that you may have heard of is XML, but that's quickly going out of favor for JSON.

There is 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 (http://www.sagtau.com/quicklookjson.html). You do need to save the output from the server to a file first:



A more readable version of the output from the web service

That makes a lot more sense. I don't know of any plug-ins for Safari that can prettify JSON directly inside the browser but other browsers such as Firefox do have such plug-ins available. (You can also find 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.)

Browse through these search results 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. When you build the search into the app, you'll store some of these fields in the `SearchResult` class so you can display them on the screen.

You might get different results from the iTunes store when you perform this search. 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 will get back another set of 50 results.

Also notice that some of these fields, such as "artistViewUrl" and "artworkUrl60" and "previewUrl" are links (URLs). For example, from the search result for the "One" music video:

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

artworkUrl60:
http://a5.mzstatic.com/us/r1000/008/Music/76/e9/43/mzi.xootxofi.80x60-
75.jpg

previewUrl:
http://a251.v.phobos.apple.com/us/r1000/040/Video/b2/e0/68/mzm.spuzitwc.
.640x480.h264lc.u.p.m4v
```

Go ahead and copy-paste these URLs in your browser. The `artistViewUrl` will open a page in the iTunes application, the `artworkUrl60` loads a thumbnail image, and the `previewUrl` opens a 50-second video preview.

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

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

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

You can add other parameters as well to make the search more specific, for example:

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

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

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

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

This searches for all apps that have something to do with angry birds. The fields in the search results for this particular query are slightly different than before. There is no more `previewUrl` but there are several screenshot URLs per entry. Different kinds of products – songs, movies, software – return different types of data.

And that's about it. All you have to do in the app is construct a URL to itunes.apple.com that has your search parameters and then use that URL to make an HTTP request. The server will send JSON gobbledygook back to the app and you'll have to somehow turn that into `SearchResult` objects and put them in the table view.

Sending the HTTP request to the iTunes server

Synchronous networking = bad

Before you begin, I should point out that there is a bad way to do networking in your apps and a good way. The bad way is to perform the HTTP requests on your app's **main thread**. This is simple to program but it will block your user interface and make your app unresponsive while the networking is taking place.

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 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) makes your apps much more responsive, but it also brings with it additional complexity that you need to deal with.

The to-do list for this section:

- Create the URL with the search parameters.
- Do the request on the iTunes server and see if you get any data back.
- Turn the JSON data into something more useful, i.e. `SearchResult` objects.
- Show these `SearchResult` objects in the table view.
- Take care of errors. There may be no network connection (or a very bad one) or the iTunes server may send back data that the app does not know how to interpret. The app should be able to recover from such situations.

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

➤ In **SearchViewController.m**, change `searchBarSearchButtonClicked:` to:

```
- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
  if ([searchBar.text length] > 0) {
    [searchBar resignFirstResponder];

    _searchResults = [NSMutableArray arrayWithCapacity:10];

    NSURL *url = [self urlWithSearchText:searchBar.text];
    NSLog(@"URL '%@'", url);

    [self.tableView reloadData];
  }
}
```

You've removed the code that creates the fake `SearchResult` items and instead call a new method, `urlWithSearchText:`. For testing purposes you log the `NSURL` object that this method returns. Note that you don't do any of this unless the user actually typed text into the search bar – it doesn't make much sense to search the iTunes store for "nothing".

➤ Add the new method below `searchBarSearchButtonClicked:`

```
- (NSURL *)urlWithSearchText:(NSString *)searchText
{
  NSString *urlString = [NSString stringWithFormat:
        @"http://itunes.apple.com/search?term=%@", searchText];
  NSURL *url = [NSURL URLWithString:urlString];
  return url;
}
```

This first uses `stringWithFormat` to build the URL as a string by placing the text from the search bar behind the "term=" parameter, and then turns this string into an `NSURL` object.

➤ Run the app and type in some search text, for example "metallica" (or one of your other favorite metal bands), and press the Search button. Xcode should now show this in its Debug pane:

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

That looks good.

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

```
URL '(null)'
```

Whoops, the app did not create a valid `NSURL` object this time. 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, `NSString` can do this encoding already, so you only have to add one extra line to the app to make this work:

```
- (NSURL *)urlWithSearchText:(NSString *)searchText
{
  NSString *escapedSearchText = [searchText
                stringByAddingPercentEscapesUsingEncoding:
                                        NSUTF8StringEncoding];

  NSString *urlString = [NSString stringWithFormat:
       @"http://itunes.apple.com/search?term=%@",
                                                escapedSearchText];
  NSURL *url = [NSURL URLWithString:urlString];
  return url;
}
```

You call the `stringByAddingPercentEscapesUsingEncoding:` method to escape the special characters, which returns a new string that you then use for the search term.

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

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

Now that you have an `NSURL` object, you can do some actual networking!

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

```
NSString *jsonString = [self performStoreRequestWithURL:url];
NSLog(@"Received JSON string '%@'", jsonString);
```

This invokes a new method, `performStoreRequestWithURL:`, which takes the `NSURL` object as a parameter and returns the JSON data that is received from the server.

➤ Add that new method:

```
- (NSString *)performStoreRequestWithURL:(NSURL *)url
{
  NSError *error;
  NSString *resultString = [NSString stringWithContentsOfURL:url
                    encoding:NSUTF8StringEncoding error:&error];
  if (resultString == nil) {
    NSLog(@"Download Error: %@", error);
    return nil;
  }
  return resultString;
}
```

The meat of this method is the call to `[NSString stringWithContentsOfURL: encoding:error:]`, a convenience constructor of the `NSString` class that returns a new string object with the data that it receives from the server at the other end of the URL. If something goes wrong, the string is `nil` and the `NSError` variable contains more details about the error.

If everything goes according to plan, then this method returns a new string object that contains the JSON data that you're after. Let's try it out!

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

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

Received JSON string '
```

```
{
 "resultCount":50,
 "results": [
{"wrapperType":"audiobook", "artistId":193508273,
"collectionId":348787393, "artistName":"William Irwin",
"collectionName":"Metallica and Philosophy: A Crash Course In Brain
Surgery",

. . . and so on . . .
```

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

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

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

```
Download Error: Error Domain=NSCocoaErrorDomain Code=256 "The operation
couldn't be completed. (Cocoa error 256.)" UserInfo=0x6c687e0
{NSURL=http://itunes.apple.com/search?term=metallica}
```

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

Parsing JSON

Now that you have managed to download a chunk of JSON data from the server, what do you do with it? JSON is a so-called *structured* data format. It typically contains arrays and dictionaries – that you know as `NSArray` and `NSDictionary` – that contain other arrays and dictionaries, as well as regular data such as string and numbers.

The JSON from the iTunes store roughly looks like this:

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

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

The `results` key contains an array, which is delineated by the `[ ]` brackets. Inside that array are more dictionaries, each of which describes a single product from the store. You can tell these things are dictionaries because they start with the `{` bracket. Here are two of these items from the array:

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

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

To summarize, the JSON data represents a dictionary, 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 an `NSString`, which isn't very handy, but using a so-called **JSON parser** you can turn this data into actual `NSDictionary` and `NSArray` objects.

JSON vs. XML

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

```
<?xml version="1.0" encoding="utf-8"?>
<iTunesSearch>
  <resultCount>5</resultCount>
  <results>
    <song>
      <artistName>Metallica</artistName>
      <trackName>Enter Sandman</trackName>
    </song>
    <song>
```

```
      <artistName>Metallica</artistName>
      <trackName>Nothing Else Matters</trackName>
    </song>

    . . . and so on . . .

  </results>
</iTunesSearch>
```

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

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

➤ Add the following method somewhere below the search bar delegate stuff:

```
- (NSDictionary *)parseJSON:(NSString *)jsonString
{
  NSData *data = [jsonString
                        dataUsingEncoding:NSUTF8StringEncoding];

  NSError *error;
  id resultObject = [NSJSONSerialization JSONObjectWithData:data
                              options:kNilOptions error:&error];
  if (resultObject == nil) {
    NSLog(@"JSON Error: %@", error);
    return nil;
  }

  return resultObject;
}
```

You're using the `NSJSONSerialization` class here to convert the JSON search results to an `NSDictionary`. Because the JSON data is actually in the form of a string, you have to put it into an `NSData` object first.

➤ Add the following lines to `searchBarSearchButtonClicked:`, before the line that reloads the table view:

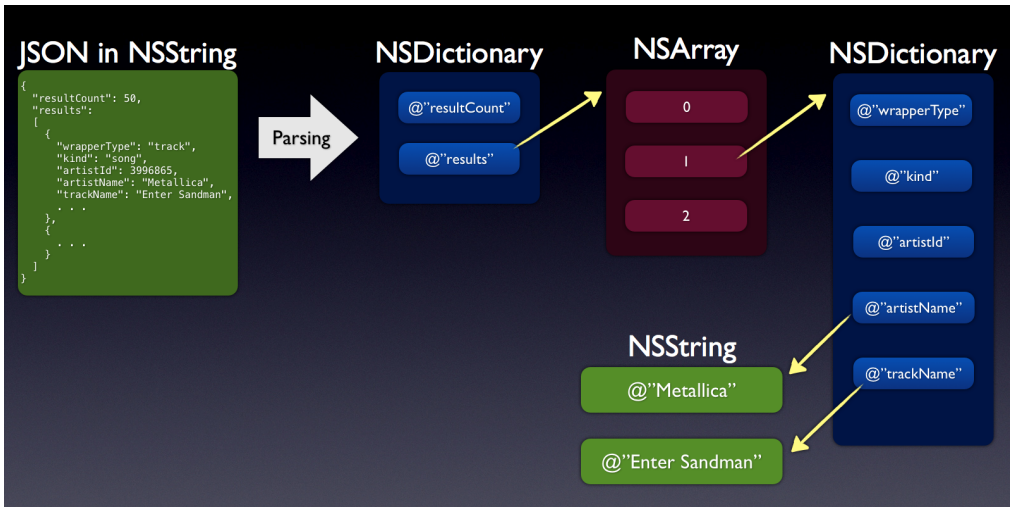
```
NSDictionary *dictionary = [self parseJSON:jsonString];
NSLog(@"Dictionary '%@'", dictionary);
```

You simply call the new `parseJSON:` method and `NSLog()` its return value.

➤ Run the app. The Xcode Debug pane now also prints the following:

```
Dictionary '{
    resultCount = 50;
    results =     (
                {
            artistId = 3996865;
            artistName = Metallica;
            kind = "music-video";
            trackName = One;

                                                . . . more fields . . .
        },
                {
            artistId = 3996865;
            artistName = Metallica;
            kind = "music-video";
            trackName = "Enter Sandman";

                                                . . . more fields . . .
        },
        . . . and so on . . .
  );
}
```

This should look very familiar to the JSON data – which is not so strange because it represents the exact same thing – except that now you're looking at the contents of an `NSDictionary` object. You have converted a bunch of text that was all packed together in a single string into actual objects that you can use.



Parsing JSON turns text into objects

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, `NSJSONSerialization` will not be able to parse the data and the app will return `nil` from `parseJSON:`.

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

You'll add some simple checks to the app to make sure you get back what you expect. If you don't own the servers you're talking to, it's best to program defensively.

▸ Add the following lines to the `parseJSON:` method:

```
- (NSDictionary *)parseJSON:(NSString *)jsonString
{
  . . .

  if (![resultObject isKindOfClass:[NSDictionary class]]) {
    NSLog(@"JSON Error: Expected dictionary");
    return nil;
  }

  return resultObject;
}
```

Just because `NSJSONSerialization` was able to turn the string into valid Objective-C objects, doesn't mean that it returns an `NSDictionary`! It could have returned an `NSArray` or even an `NSString` or `NSNumber`... In the case of the iTunes store web service, the top-level object *should* be an `NSDictionary`, 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 an `NSDictionary` but an `NSArray`.

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 check that the object returned by `NSJSONSerialization` is truly an `NSDictionary` and if it's not, you return `nil` to signal an error.

Speaking of errors, let's add some alert views.

▸ Add the following method below `parseJSON:`

```
- (void)showNetworkError
```

```
{
  UIAlertView *alertView = [[UIAlertView alloc]
    initWithTitle:@"Whoops..."
    message:@"There was an error reading from the iTunes Store. Please
try again."
    delegate:nil
    cancelButtonTitle:@"OK"
    otherButtonTitles:nil];

  [alertView show];
}
```

➤ Change `searchBarSearchButtonClicked:` to the following:

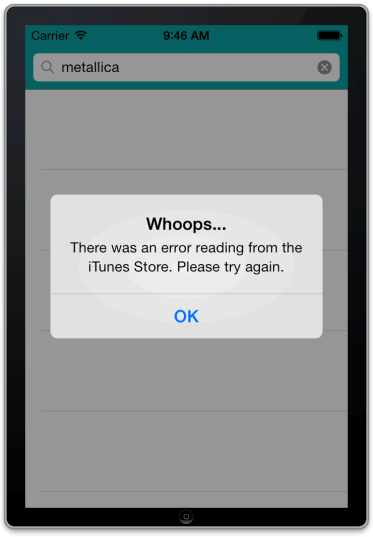
```
- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
  if ([searchBar.text length] > 0) {
    [searchBar resignFirstResponder];

    _searchResults = [NSMutableArray arrayWithCapacity:10];

    NSURL *url = [self urlWithSearchText:searchBar.text];
    NSString *jsonString = [self
                                performStoreRequestWithURL:url];
    if (jsonString == nil) {
      [self showNetworkError];
      return;
    }

    NSDictionary *dictionary = [self parseJSON:jsonString];
    if (dictionary == nil) {
      [self showNetworkError];
      return;
    }

    NSLog(@"Dictionary '%@'", dictionary);

    [self.tableView reloadData];
  }
}
```

If something goes wrong, you call the `showNetworkError` method to show an alert box.

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

➤ Try this: In the `urlWithSearchText:` method, temporarily change the itunes.apple.com part of the URL to "NOMOREitunes.apple.com". You should now get an error alert when you try a search because no such server exists at that address. This simulates the iTunes server being down. Don't forget to change the URL back when you're done testing.

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



The app shows an alert when there is a network error

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

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

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

So who was right? It could have been a bug in `NSJSONSerialization` or it could be that the iTunes web service did something naughty... As of the current revision of this tutorial, searching for "photoshop" works again. In any case, it should be

obvious that when you're doing networking things can – and will! – go wrong, often in unexpected ways.

Turning the JSON into SearchResult objects

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

The iTunes store sells different kinds of products – songs, ebooks, 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 and podcasts
- Audio books
- Software (apps)
- E-books

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

➤ First, add a new method call to `searchBarSearchButtonClicked:`, just before the line that reloads the table view.

```
  [self parseDictionary:dictionary];
```

This new method goes through the top-level `NSDictionary` and looks at each search result in turn. If it's a type of product the app supports, then it creates a `SearchResult` object for that product and adds it to the `searchResults` array.

➤ Add the `parseDictionary:` method below `parseJSON:`

```
- (void)parseDictionary:(NSDictionary *)dictionary
{
  NSArray *array = dictionary[@"results"];
  if (array == nil) {
    NSLog(@"Expected 'results' array");
    return;
  }

  for (NSDictionary *resultDict in array) {
```

```
    NSLog(@"wrapperType: %@, kind: %@",
                resultDict[@"wrapperType"], resultDict[@"kind"]);
  }
}
```

First there is another bit of defensive programming. This makes sure the dictionary contains a key named `results` that contains an `NSArray`. It probably will, but better safe than sorry. Once it is satisfied that `NSArray` exists, the method uses a for-loop to look at each of the array's elements in turn. Remember that each of the elements from the array is another `NSDictionary`. For each of these dictionaries, you print out the value of its `wrapperType` and `kind` fields.

➤ 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.

```
StoreSearch[34492:f803] wrapperType: track, kind: feature-movie
StoreSearch[34492:f803] wrapperType: track, kind: music-video
StoreSearch[34492:f803] wrapperType: track, kind: song
```

To turn these things into `SearchResult` objects, you're going to look at value of the `wrapperType` field first. If that is "track" then you know that the product in question is a song, movie, music video, podcast or episode of a TV show. Other values for `wrapperType` are "audiobook" for audio books and "software" for apps, and you will interpret these differently than "tracks". But before you get to that, let's first add some new properties to the `SearchResult` object.

Documentation!

If you're wondering how I knew how to interpret the data from the iTunes web service, or even how to make the URLs to use the service in the first place, then you should know 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:

http://www.apple.com/itunes/affiliates/resources/documentation/itunes-store-web-service-search-api.html

Just reading the docs is often not enough. You have to play with the web service for a bit to know what you can and cannot do. There are some things that the StoreSearch app needs to do with the search results that were not clear from reading the documentation – for example, ebooks do not include a `wrapperType` field for some reason. So first read the docs and then play with it.

That goes for any API, really, whether it's something from the iOS SDK or a web service.

A better SearchResult

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

➤ Add the following properties to **SearchResult.h**:

```
@property (nonatomic, copy) NSString *artworkURL60;
@property (nonatomic, copy) NSString *artworkURL100;
@property (nonatomic, copy) NSString *storeURL;
@property (nonatomic, copy) NSString *kind;
@property (nonatomic, copy) NSString *currency;
@property (nonatomic, copy) NSDecimalNumber *price;
@property (nonatomic, copy) NSString *genre;
```

You're not including everything that the iTunes store returns, only the fields that you're interested in. `SearchResult` stores two artwork URLs, one for a 60×60 pixel image and the other for a 100×100 pixel image. It alsos store the kind and genre of the item, its price and the currency (US dollar, Euro, British Pounds, etc.), as well as a link to the product's page on the iTunes store itself.

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

➤ Back in **SearchViewController.m**, change the for-loop in `parseDictionary:` to the following:

```
  for (NSDictionary *resultDict in array) {

    SearchResult *searchResult;

    NSString *wrapperType = resultDict[@"wrapperType"];

    if ([wrapperType isEqualToString:@"track"]) {
      searchResult = [self parseTrack:resultDict];
    }

    if (searchResult != nil) {
      [_searchResults addObject:searchResult];
    }
  }
```

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

➤ Add the `parseTrack:` method below `parseDictionary:`

```
- (SearchResult *)parseTrack:(NSDictionary *)dictionary
{
  SearchResult *searchResult = [[SearchResult alloc] init];
  searchResult.name = dictionary[@"trackName"];
  searchResult.artistName = dictionary[@"artistName"];
  searchResult.artworkURL60 = dictionary[@"artworkUrl60"];
  searchResult.artworkURL100 = dictionary[@"artworkUrl100"];
  searchResult.storeURL = dictionary[@"trackViewUrl"];
  searchResult.kind = dictionary[@"kind"];
  searchResult.price = dictionary[@"trackPrice"];
  searchResult.currency = dictionary[@"currency"];
  searchResult.genre = dictionary[@"primaryGenreName"];
  return searchResult;
}
```

What happens here is quite simple. You first allocate a new `SearchResult` object and then get the values out of the dictionary and put them in the `SearchResult`'s properties.

All of these things are strings, except `price`, which is an `NSDecimalNumber`. You haven't seen this type before but it's a subclass of `NSNumber` that allows for doing high precision calculations with numbers that have decimal points, such as money. When it parses the JSON data, `NSJSONSerialization` detects that the `price` field contains numbers and so it creates an `NSDecimalNumber` object.

Note: The notation for retrieving an object from a dictionary looks very similar to indexing an array:

```
object = array[index]

object = dictionary[@"key"]
```

The difference between the two is that for indexing an array you always use an integer but for a dictionary you typically use a string.

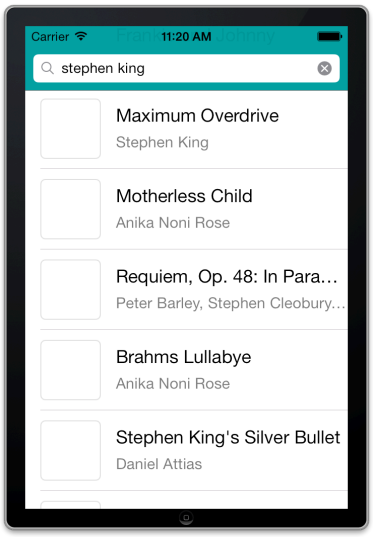
The old notation for obtaining a value from a dictionary is:

```
object = [dictionary objectForKey:@"key"]
```

You will occasionally see this in older code but the `[]` notation is more common nowadays.

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

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



The results from the search now show up in the table

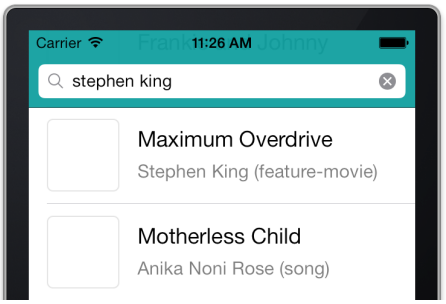
You might be wondering what "Brahms Lullabye" and "Requiem, Op. 48" have to do with Stephen King (they're not new novels!). The search results may include podcasts, songs, or other related products. It would be useful to make the table view display what type of product it is showing, so let's improve `cellForRowAtIndexPath` a little.

▸ Change `cellForRowAtIndexPath` to the following:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
        cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  if ([_searchResults count] == 0) {
    return [tableView dequeueReusableCellWithIdentifier:
              NothingFoundCellIdentifier forIndexPath:indexPath];
  } else {
    SearchResultCell *cell = (SearchResultCell *)[tableView
        dequeueReusableCellWithIdentifier:
              SearchResultCellIdentifier forIndexPath:indexPath];

    SearchResult *searchResult = _searchResults[indexPath.row];
    cell.nameLabel.text = searchResult.name;

    NSString *artistName = searchResult.artistName;
```

```
    if (artistName == nil) {
      artistName = @"Unknown";
    }

    NSString *kind = searchResult.kind;
    cell.artistNameLabel.text = [NSString stringWithFormat:
                                 @"%@ (%@)", artistName, kind];

    return cell;
  }
}
```

The first change here is that you now check that the `SearchResult`'s `artistName` is not `nil`. When testing the app I noticed that sometimes a search result did not include an artist name. In that case you make the cell say "Unknown". You also add the value of the `kind` property to the artist name label, which should tell the user what kind of product they're looking at:



Yup, they're not books

There is one problem with this. The value of `kind` comes straight from the server and it is more of an internal name than something you'd want to show directly to the user. What if you want it to say "Movie" instead, or maybe you want to translate the app to another language (something you'll do later in this tutorial). It's better to convert this internal identifier ("feature-movie") into the text that you want to show to the user ("Movie").

➤ Add this new method below `cellForRowAtIndexPath:`

```
- (NSString *)kindForDisplay:(NSString *)kind
{
  if ([kind isEqualToString:@"album"]) {
    return @"Album";
  } else if ([kind isEqualToString:@"audiobook"]) {
    return @"Audio Book";
  } else if ([kind isEqualToString:@"book"]) {
    return @"Book";
  } else if ([kind isEqualToString:@"ebook"]) {
```

```
    return @"E-Book";
  } else if ([kind isEqualToString:@"feature-movie"]) {
    return @"Movie";
  } else if ([kind isEqualToString:@"music-video"]) {
    return @"Music Video";
  } else if ([kind isEqualToString:@"podcast"]) {
    return @"Podcast";
  } else if ([kind isEqualToString:@"software"]) {
    return @"App";
  } else if ([kind isEqualToString:@"song"]) {
    return @"Song";
  } else if ([kind isEqualToString:@"tv-episode"]) {
    return @"TV Episode";
  } else {
    return kind;
  }
}
```

These are the types of products that this app understands. It's possible that I missed one or that the iTunes Store adds a new product type at some point. In that case you'll simply return the original `kind` value (and hopefully fix this in an update of the app).

➤ In `cellForRowAtIndexPath`, change the line,

```
NSString *kind = searchResult.kind;
```

to:

```
NSString *kind = [self kindForDisplay:searchResult.kind];
```

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

The product type is a bit more human-friendly

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

➤ Add the following methods below `parseTrack:`

```
- (SearchResult *)parseAudioBook:(NSDictionary *)dictionary
{
  SearchResult *searchResult = [[SearchResult alloc] init];
  searchResult.name = dictionary[@"collectionName"];
  searchResult.artistName = dictionary[@"artistName"];
  searchResult.artworkURL60 = dictionary[@"artworkUrl60"];
  searchResult.artworkURL100 = dictionary[@"artworkUrl100"];
  searchResult.storeURL = dictionary[@"collectionViewUrl"];
  searchResult.kind = @"audiobook";
  searchResult.price = dictionary[@"collectionPrice"];
  searchResult.currency = dictionary[@"currency"];
  searchResult.genre = dictionary[@"primaryGenreName"];
  return searchResult;
}

- (SearchResult *)parseSoftware:(NSDictionary *)dictionary
{
  SearchResult *searchResult = [[SearchResult alloc] init];
  searchResult.name = dictionary[@"trackName"];
  searchResult.artistName = dictionary[@"artistName"];
  searchResult.artworkURL60 = dictionary[@"artworkUrl60"];
  searchResult.artworkURL100 = dictionary[@"artworkUrl100"];
  searchResult.storeURL = dictionary[@"trackViewUrl"];
  searchResult.kind = dictionary[@"kind"];
  searchResult.price = dictionary[@"price"];
  searchResult.currency = dictionary[@"currency"];
  searchResult.genre = dictionary[@"primaryGenreName"];
  return searchResult;
}

- (SearchResult *)parseEBook:(NSDictionary *)dictionary
{
  SearchResult *searchResult = [[SearchResult alloc] init];
  searchResult.name = dictionary[@"trackName"];
  searchResult.artistName = dictionary[@"artistName"];
  searchResult.artworkURL60 = dictionary[@"artworkUrl60"];
  searchResult.artworkURL100 = dictionary[@"artworkUrl100"];
  searchResult.storeURL = dictionary[@"trackViewUrl"];
  searchResult.kind = dictionary[@"kind"];
  searchResult.price = dictionary[@"price"];
  searchResult.currency = dictionary[@"currency"];
  searchResult.genre = [(NSArray *)dictionary[@"genres"]
                                componentsJoinedByString:@", "];
  return searchResult;
}
```

Two interesting points here: Audio books don't have a "kind" field, so you have to set the `kind` property to `@"audiobook"` yourself.

E-books don't have a "primaryGenreName" field, but an array of genres. You use the `componentsJoinedByString` method from `NSArray` to glue these genre names into a single string, separated by commas.

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

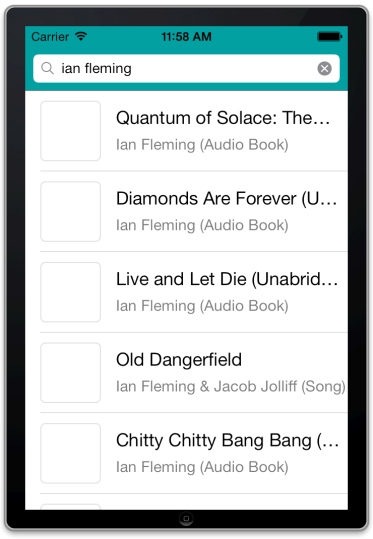
▸ Change the for-loop in `parseDictionary:` to:

```
  for (NSDictionary *resultDict in array) {

    SearchResult *searchResult;

    NSString *wrapperType = resultDict[@"wrapperType"];
    NSString *kind = resultDict[@"kind"];

    if ([wrapperType isEqualToString:@"track"]) {
      searchResult = [self parseTrack:resultDict];
    } else if ([wrapperType isEqualToString:@"audiobook"]) {
      searchResult = [self parseAudioBook:resultDict];
    } else if ([wrapperType isEqualToString:@"software"]) {
      searchResult = [self parseSoftware:resultDict];
    } else if ([kind isEqualToString:@"ebook"]) {
      searchResult = [self parseEBook:resultDict];
    }

    if (searchResult != nil) {
      [_searchResults addObject:searchResult];
    }
  }
```

For some reason, e-books do not have a `wrapperType` field, so in order to determine whether something is an e-book you have to look at the `kind` field instead. If there is a `wrapperType` or `kind` that the app does not support, then no `SearchResult` object gets created, the value of `searchResult` is `nil`, and you simply skip that item.

▸ Run the app and search for some software, audio books or e-books. It can take a few tries before you find some because of the enormous quantity of products on the store.

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



The app shows a varied range of products now

One more thing, it would be nice to sort the search results alphabetically. That's quite easy, actually. `NSMutableArray` already has a method to sort itself. All you have to do is tell it what to sort on.

▸ Change the bottom of `searchBarSearchButtonClicked:` to:

```
- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
  if ([searchBar.text length] > 0) {
    . . .

    [self parseDictionary:dictionary];
    [_searchResults sortUsingSelector:@selector(compareName:)];

    [self.tableView reloadData];
  }
}
```

Before reloading the table, you first call `sortUsingSelector:` on the `_searchResults` array. The selector is a method on `SearchResult` that you still have to define.

▸ Add the following method signature to **SearchResult.h**:

```
- (NSComparisonResult)compareName:(SearchResult *)other;
```

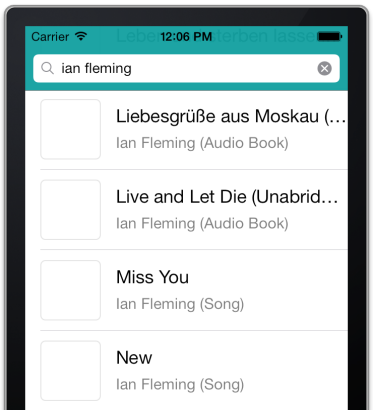
▸ And its implementation in **SearchResult.m**:

```
- (NSComparisonResult)compareName:(SearchResult *)other
{
  return [self.name localizedStandardCompare:other.name];
```

```
}
```

You have done this before in one of the other tutorials. In order to sort its contents, the `_searchResults` array will compare the `SearchResult` objects with each other and put them into an ordered fashion. It does this by calling the `compareName:` method, which simply compares the name of one `SearchResult` object to the name of another using `NSString`'s `localizedStandardCompare:` method.

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



The search results are sorted by name

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

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 just wanted to show 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: http://www.apple.com/itunes/affiliates/resources/documentation/itunes-store-web-service-search-api.html

➤ Commit your changes.

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

SDKs for APIs

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

For example: Facebook (https://developers.facebook.com/docs/ios/ios-sdk-tutorial/), Wordnik (http://developer.wordnik.com), Amazon Web Services (http://aws.amazon.com/sdkforios/), and many others. If you're really lucky, support for the web service is already built into iOS itself, such as the Social Framework that makes it very easy to put Twitter and Facebook into your apps.

Asynchronous networking

That wasn't so bad, or was it? Yes it was, and I'll show you why! Did you notice that whenever you performed a search, the app became unresponsive? While the network request was taking place, you could not scroll the table view up or down, or type anything new into the search bar. The app was completely frozen for a few seconds.

You may not have seen this if your network connection was very fast but if you're using your iPhone out in the wild the network will be a lot slower than your home or office Wi-Fi, and a search can easily take ten seconds or more. Maybe you're not so worried that the app is unresponsive while the search is taking place. After all, there is nothing for the user to do at that point anyway...

However, to most users an app that does not respond is an app that has crashed. The screen looks empty, there is no indication of what is going on, and even an innocuous gesture such as sliding your finger up and down does not bounce the table view like you'd expect it to. Conclusion: the app has crashed. The user will press the Home button and try again – or worse, delete your app, give it a bad rating on the App Store, and switch to a competing app.

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

First off, let's increase the amount of data that the app will get back. You can add a "limit" parameter to the URL that sets the maximum number of results that the web service will return to you. The default value is 50, the maximum is 200.

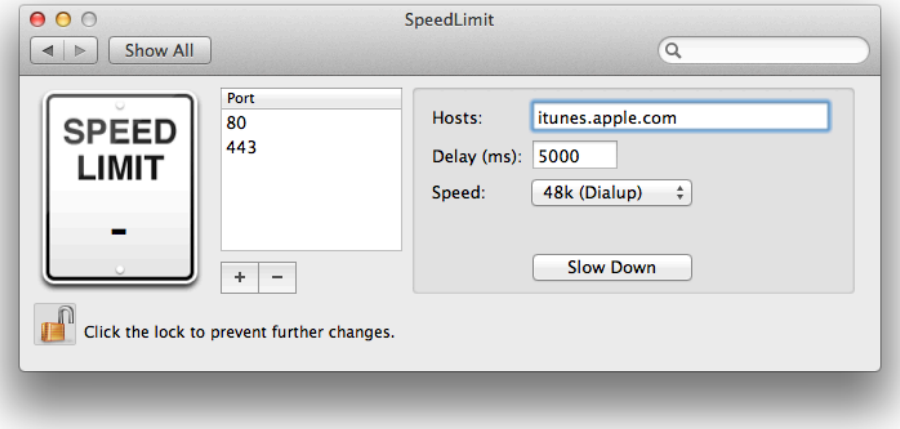
➤ In `urlWithSearchText:`, change the following line:

```
NSString *urlString = [NSString stringWithFormat:
       @"http://itunes.apple.com/search?term=%@&limit=200",
       escapedSearchText];
```

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

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

Still too fast? Then download and install the SpeedLimit preference pane (http://mschrag.github.com/). This adds a new pane to your System Preferences window that lets you slow down network connections to specific addresses.



Setting up the SpeedLimit preferences pane

➤ Open the **System Preferences** on your Mac, switch to the **SpeedLimit** panel, and click the lock icon to make changes. In the Hosts field, type **itunes.apple.com**. Set the delay to **5000** milliseconds (i.e. 5 seconds) and the speed to **48K (Dialup)**. Then press the **Slow Down** button.

➤ Now run the app and search for something. The SpeedLimit tool will delay the actual outgoing HTTP request by 5 seconds in order to simulate a slow connection, and then downloads the data at a very slow speed. (If the download still appears very fast, then try searching for some term you haven't used before; the system may be caching the results from a previous search.)

Notice how the app totally doesn't respond during this time? It feels like something is wrong. Did the app crash or is it still doing something? It's impossible to tell and very confusing to your users when this happens. (Even worse, if your program is unresponsive for too long, iOS may actually kill it by force, in which case it really did crash. You don't want that to happen!)

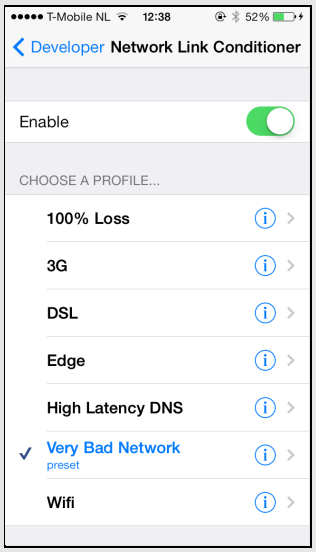
"Ah," you say, "let's show some type of animation to let the user know that the app is communicating with a server. Then at least they will know that the app is busy." That sounds like a decent thing to do, so let's get to it.

Tip: Another great tool to fake bad network connections is Network Link Conditioner. This is also a preferences pane and can be used to simulate different network conditions, including bad cell phone networks.

If your System Preferences does not include Network Link Conditioner yet, then you first need to install it. From Xcode's menu choose **Open Developer**

Tool → More Developer Tools... This opens the Apple developer website. From the Downloads for Apple Developers page, download the latest **Hardware IO Tools for Xcode** package. Open the downloaded **.dmg** file and double-click **Network Link Conditioner.prefPane** to install it.

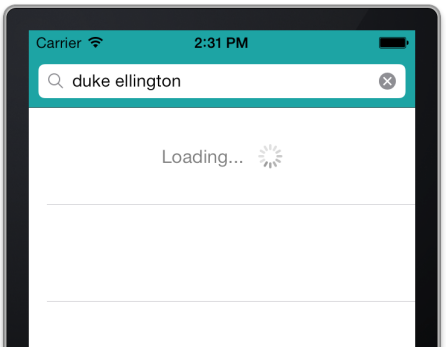
Even better than pretending to have a lousy connection on the Simulator is to use Network Link Conditioner on your device, so you can also test bad network connections on your actual iPhone. You can find it under **Settings → Developer → Network Link Conditioner**:



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 a broadband connection…

The activity indicator

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



The app shows that it is busy

➤ Create a new, empty nib file. Call it **LoadingCell.xib**.

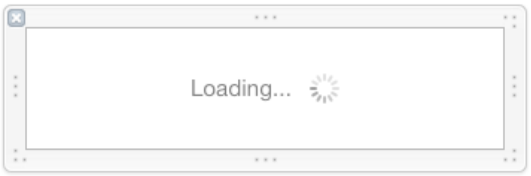
➤ Drag a new **Table View Cell** into the canvas. Set its height to 80 points.

➤ 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 15.0**. 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. Change its **Style** to **Gray** and give it the **Tag** 100.

The final design looks like this:



The design of the LoadingCell nib

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 top of **SearchViewController.m**:

```
static NSString * const LoadingCellIdentifier = @"LoadingCell";
```

➤ And register the nib in `viewDidLoad`:

```
cellNib = [UINib nibWithNibName:LoadingCellIdentifier
                         bundle:nil];

[self.tableView registerNib:cellNib
     forCellReuseIdentifier:LoadingCellIdentifier];
```

Now you have to come up with some way to let the table view's data source know that the app is currently in a state of downloading data from the server. The simplest way to do that is to keep a boolean flag. If this `BOOL` is `YES`, then the app is downloading stuff and the new Loading... cell should be shown; if the `BOOL` is `NO`, then you show the regular contents of the table view.

➤ Add a new instance variable:

```
@implementation SearchViewController
{
  NSMutableArray *_searchResults;
  BOOL _isLoading;
```

```
}
```

➤ Change `numberOfRowsInSection` to:

```
- (NSInteger)tableView:(UITableView *)tableView
             numberOfRowsInSection:(NSInteger)section
{
  if (_isLoading) {
    return 1;
  } else if (_searchResults == nil) {
    return 0;
  } else if ([_searchResults count] == 0) {
    return 1;
  } else {
    return [_searchResults count];
  }
}
```

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

➤ Change `cellForRowAtIndexPath` to:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  if (_isLoading) {
    UITableViewCell *cell = [tableView
         dequeueReusableCellWithIdentifier:LoadingCellIdentifier
         forIndexPath:indexPath];

    UIActivityIndicatorView *spinner =
              (UIActivityIndicatorView *)[cell viewWithTag:100];
    [spinner startAnimating];

    return cell;

  } else if ([_searchResults count] == 0) {
    return [tableView dequeueReusableCellWithIdentifier:
             NothingFoundCellIdentifier forIndexPath:indexPath];

  } else {
    . . .
```

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

▸ Change `willSelectRowAtIndexPath` to:

```
- (NSIndexPath *)tableView:(UITableView *)tableView
             willSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  if ([_searchResults count] == 0 || _isLoading) {
    return nil;
  } else {
    return indexPath;
  }
}
```

Just like you don't want the users to select the "Nothing Found" cell, you also don't want them to select the "Loading..." cell, so you return `nil` in both cases.

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

▸ Change `searchBarSearchButtonClicked:` to:

```
- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
  if ([searchBar.text length] > 0) {
    [searchBar resignFirstResponder];

    _isLoading = YES;
    [self.tableView reloadData];

    . . . here is the networking code . . .

    _isLoading = NO;
    [self.tableView reloadData];
  }
}
```

Before you do the networking request, you set `_isLoading` to `YES` and reload the table to show the activity indicator. After the request completes, you set `_isLoading` back to `NO` and reload the table again to show the search results.

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

▸ Run the app and perform a search. While search is taking place the Loading... cell with the spinning activity indicator should appear... or should it?!

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

➤ To show you why, first change `searchBarSearchButtonClicked:` to the following and run the app again.

```
- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
  if ([searchBar.text length] > 0) {
    [searchBar resignFirstResponder];

    _isLoading = YES;
    [self.tableView reloadData];
  }
}
```

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

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

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

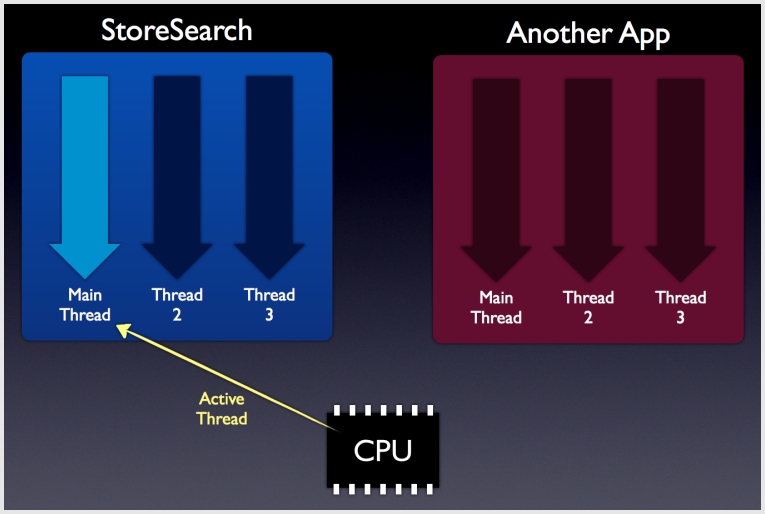
The main thread

The CPU (Central Processing Unit) in older iPhone and iPad models has only one core, which means it can only do one thing at the time. More recent models have a CPU with two cores, which allows two simultaneous computations to happen. But that's still not very much. Modern computers often have many processes running at the same time. To get around this hardware limitation most computers, including the iPhone and iPad, use **preemptive multitasking** and **multithreading** to give the illusion that they can do many things at once.

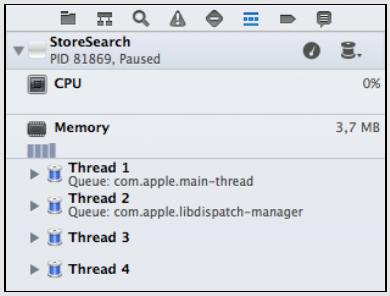
Multitasking is something that happens between different apps. Each app is said to have its own **process** and each process is given a small portion of each second of CPU time to perform its jobs. Then it is *pre-empted* and control is given to the next process. Apple only introduced multitasking between apps with iOS 4 but in truth all iPhones since version 1.0 have been running true multitasked processes – you simply weren't able to take advantage of that as a user until iOS 4.

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



Most of these threads are managed by iOS itself and you don't have to worry about them. 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 "block" the main thread. If your action method takes more than a fraction of a second to run, then doing all these computations on the main thread is not a good idea for the reasons you saw earlier: the app becomes unresponsive because the main thread cannot handle any UI events while you're keeping it busy doing something else – and if the operation takes too long the app may even be killed by the system.

In StoreSearch, you're doing a lengthy network operation on the main thread. It could potentially take many seconds, maybe even minutes, to complete. After you set the `_isLoading` flag to `YES`, 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 never gets around to handling that event because you immediately keep the thread busy with the networking operation.

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

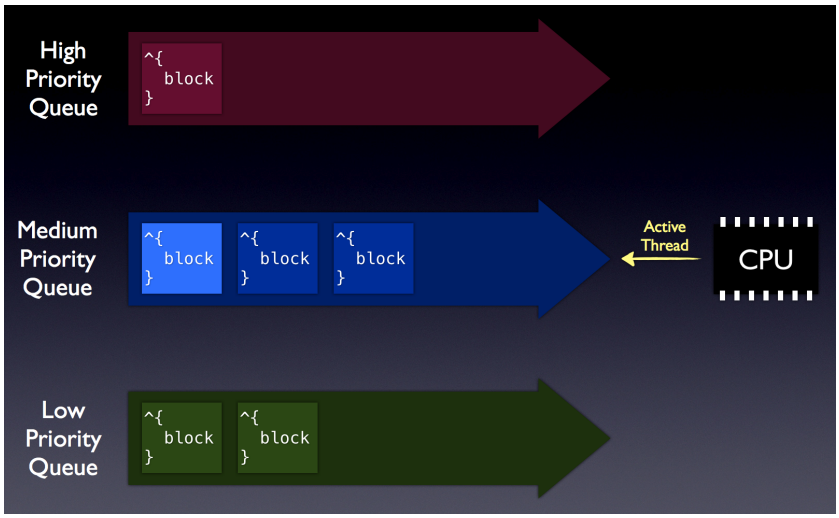
Making it asynchronous

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

That is not to say you should create your own thread. If you've programmed on other platforms before you may not think twice about creating new threads, but on iOS that is often not the best solution. You see, threads are tricky. Not threads per se, but doing things in parallel. Our human minds are very bad at handling the complexity that comes from doing more than one thing at a time – at least when it comes to computations. I won't go into too much detail here, but generally you want to avoid the situation where two threads are modifying the same piece of data at the same time – that can lead to very surprising (but not very pleasant!) results.

Rather than making your own threads, iOS has several more convenient ways to start background processes. For this app you'll be using **queues** and **Grand Central Dispatch** (or GCD). You've already briefly played with **blocks** when you did reverse geocoding in the MyLocations tutorial. Blocks and GCD were added to iOS 4 and greatly simplify certain programming tasks.

In short, GCD has a number of queues and you can put new blocks on each of these queues. A block contains a bit of programming code and GCD will perform these pieces of code one-by-one in the background. Exactly how it does that is not important; you're only guaranteed it happens on a background thread somewhere. Queues are not exactly the same as threads, but they use threads to do their dirty work.



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

To perform a job in the background, you put it in a block and then give that block to a queue and forget about it. It's as simple as that.

To make the web service requests asynchronous, you're going to put the networking part from `searchBarSearchButtonClicked:` into a block and then place that block on a medium priority queue.

➤ Change `searchBarSearchButtonClicked:` to the following:

```
- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
  if ([searchBar.text length] > 0) {
    [searchBar resignFirstResponder];

    _isLoading = YES;
    [self.tableView reloadData];

    _searchResults = [NSMutableArray arrayWithCapacity:10];

    dispatch_queue_t queue = dispatch_get_global_queue(
                             DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

    dispatch_async(queue, ^{

      NSURL *url = [self urlWithSearchText:searchBar.text];
      NSString *jsonString = [self
                                performStoreRequestWithURL:url];
      if (jsonString == nil) {
        NSLog(@"Error!");
        return;
      }
```

```
      NSDictionary *dictionary = [self parseJSON:jsonString];
      if (dictionary == nil) {
        NSLog(@"Error!");
        return;
      }

      [self parseDictionary:dictionary];
      [_searchResults sortUsingSelector:
                                  @selector(compareName:)];

      NSLog(@"DONE!");
    });
  }
}
```

Here is the new stuff:

```
dispatch_queue_t queue = dispatch_get_global_queue(
                            DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
```

This gets a reference to the queue. GCD uses regular functions, not classes, because it is not limited to Objective-C and can be used from C and C++ programs as well.

Once you have the queue, you can dispatch a block on it:

```
dispatch_async(queue, ^{
    // this is the block
});
```

The block is everything between the ^{ and } symbols. Whatever code is in the block will be put on the queue and is executed asynchronously in the background. After scheduling this block, the main thread is free to continue. It is no longer halted.

Notice that inside the block I have removed the code that reloads the table view after the search is done, as well as the error handling code. 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! – and after a short while you should see the "DONE!" message appear in the debug pane. Of course, the Loading... cell sticks around forever because you haven't told it yet to go away.

The reason I removed all the user interface code from the block is that UIKit has a rule that all UI code should always be performed on the main thread. This is important! Because accessing the same data from multiple threads can create all

sorts of misery, 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 block because it runs on a queue that is backed by a thread other than the main thread.

As it happens, there is also a so-called "main queue" that is associated with the main thread. If you need to do anything on the main thread from a background queue, you can simply create a new block and schedule that on the main queue. Like this:

```
dispatch_async(dispatch_get_main_queue(), ^{
  _isLoading = NO;
  [self.tableView reloadData];
});
```

The `dispatch_get_main_queue()` function returns a reference to the main queue and `dispatch_async()` then schedules a new block on that queue. It's a new block because there are `^{ }` symbols again.

➤ The final version of `searchBarSearchButtonClicked:` is:

```
- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
  if ([searchBar.text length] > 0) {
    [searchBar resignFirstResponder];

    _isLoading = YES;
    [self.tableView reloadData];

    _searchResults = [NSMutableArray arrayWithCapacity:10];

    dispatch_queue_t queue = dispatch_get_global_queue(
                             DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

    dispatch_async(queue, ^{

      NSURL *url = [self urlWithSearchText:searchBar.text];
      NSString *jsonString = [self
                                performStoreRequestWithURL:url];
      if (jsonString == nil) {
        dispatch_async(dispatch_get_main_queue(), ^{
          [self showNetworkError];
        });
        return;
      }

      NSDictionary *dictionary = [self parseJSON:jsonString];
```

```
      if (dictionary == nil) {
        dispatch_async(dispatch_get_main_queue(), ^{
          [self showNetworkError];
        });
        return;
      }

      [self parseDictionary:dictionary];
      [_searchResults sortUsingSelector:
                                  @selector(compareName:)];

      dispatch_async(dispatch_get_main_queue(), ^{
        _isLoading = NO;
        [self.tableView reloadData];
      });
    });
  }
}
```

Notice that you also schedule the calls to `[self showNetworkError]` on the main queue. That method shows a `UIAlertView`, which is UI code and therefore needs to happen on the main thread.

➤ Try it out. With those changes in place, your networking code no longer occupies the main thread and the app suddenly feels a lot more responsive!

When working with blocks you will often see this pattern:

```
dispatch_queue_t queue = dispatch_get_global_queue(
                            DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_async(queue, ^{
  // code that needs to run in the background

  dispatch_async(dispatch_get_main_queue(), ^{
    // update the user interface
  });
});
```

The syntax of blocks may look a little scary, but just remember that everything in between the `^{` and `}` markers is a block and will not be performed right there but at some later point.

➤ I think with this important improvement the app deserves a new version number, so commit the changes and create a tag for **v0.2**.

You can find the project files for this section under **04 - Async Networking** in the tutorial's Source Code folder.

AFNetworking

So far you've used `NSString`'s `stringWithContentsOfURL:` 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 Objective-C classes such as `NSURLConnection` and `NSURLSession`.

Even convenient sometimes isn't convenient enough, so third party developers have come up with packages that are even easier to use. In this section you'll replace the existing networking code with the **AFNetworking** library.

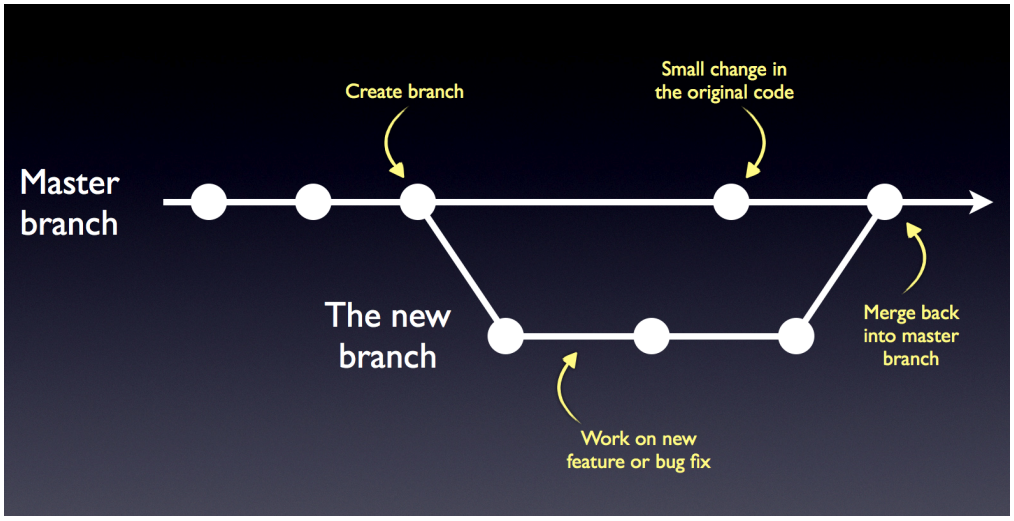
Note: Another very popular networking library is ASIHTTPRequest. You will often see it recommended on blogs and forums. It's a pretty good library with many features but it's also a bit old and is no longer being maintained. AFNetworking is a more up-to-date choice, which is why you'll be using it in this tutorial. There are several other good third-party networking libraries and I encourage you to play with them all.

Branch it

Whenever you make a big change to the code, such as adding AFNetworking, there is a possibility that you will mess things up. I certainly do often enough! That's why it's smart to create a so-called Git **branch** first.

The Git repository contains a history of all the app's code, but it can also contain this history along different paths. You just finished the first version of the networking code and it works pretty well. Now you're going to completely replace that with a – hopefully! – better solution. In doing so, you may want to commit your progress at several points along the way.

What if it turns out that switching to AFNetworking 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.

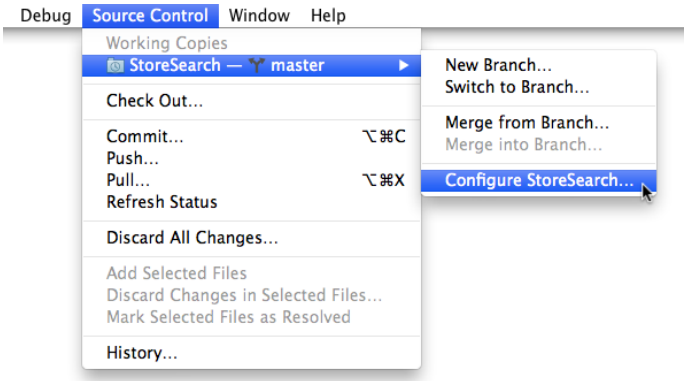


Branches in action

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 satisfied that everything works as it should, you merge your changes back into the master branch. Different people use different branching strategies but this is the general principle.

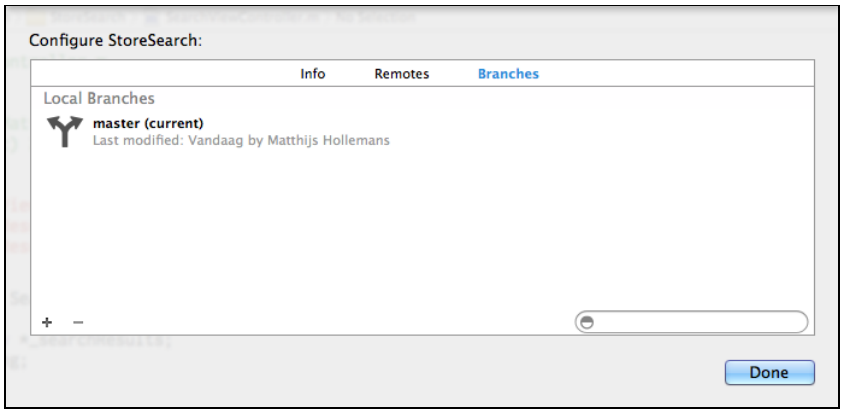
So far you have been committing your changes to the "master" branch. Now you're going to make a new branch, let's call it "AFNetworking", and commit your changes to that. When you're done with this new feature you will merge everything back into the master branch.

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



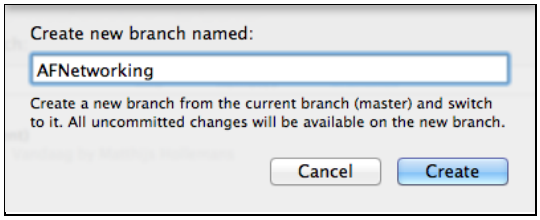
The Source Control branch menu

› Select **StoreSearch – master** (the name of the active branch), and choose **Configure StoreSearch...** to bring up the following panel:



There is currently only one branch in the repository

› Go to the **Branches** tab and click the **+** button at the bottom. In the screen that appears, type **AFNetworking** for the branch name and click **Create**.



Creating a new branch

When Xcode is done, you'll see that a new AFNetworking branch has been added and that it is made the current one. This new branch contains the exact same history as the master branch. However, from here on out the two paths will diverge. Any changes you make happen on the AFNetworking branch only.

Adding AFNetworking to the project

Even though you could write all of the code that you will ever use in your apps totally by yourself, a smart programmer will try to reuse as much code from others as possible. AFNetworking is a good example of that. My projects often make use of several reusable libraries, and in this section I'd like to show you how to integrate such third-party code into your own projects.

By far the easiest way to add third-party libraries to your projects is through **CocoaPods**.

› To install CocoaPods, open a Terminal and type the following command:

```
sudo gem install cocoapods
```

This asks for your administrator password and then it will download the required packages and install them.

› Still in Terminal, `cd` to your StoreSearch project folder:

```
cd /Users/matthijs/Desktop/StoreSearch
```

➤ Now type the following command:

```
pod init
```

This creates a new file named **Podfile** inside your project folder. Open this file in a text editor (or in Xcode) and change it to:

```
platform :ios, "7.0"

target "StoreSearch" do

pod 'AFNetworking', '~> 2.0'

end

target "StoreSearchTests" do

end
```

This tells CocoaPods that you're building an app for iOS 7.0 or better and that you want to include AFNetworking 2.0.

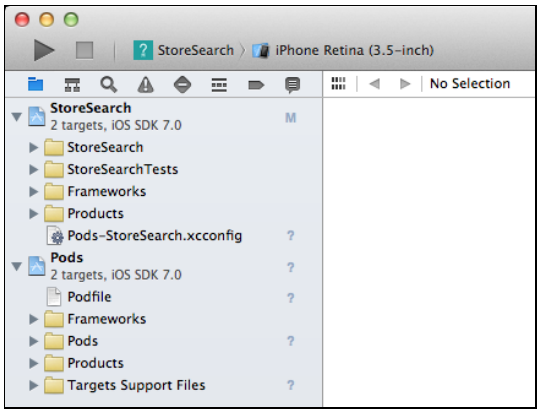
➤ Save the Podfile and go back to Terminal. Type the command:

```
pod install
```

This is what makes the magic happen. CocoaPods downloads the source files for AFNetworking and puts them into a new Xcode project named **Pods**. It also generates an Xcode *workspace*, which is a way to group two or more projects together.

➤ Close the StoreSearch project if that was still open in Xcode. From now on if you want to work on the app you no longer open StoreSearch.xcodeproj but the workspace, **StoreSearch.xcworkspace**. (Simply double-click on the file in Finder to open it.)

Xcode now lists two projects in the navigator, StoreSearch and Pods:



The workspace with the StoreSearch and Pods projects

The Pods project contains all the libraries that you specified in the Podfile. For this app that is only AFNetworking but you can add many others (there are over 2500 packages available for CocoaPods).

You typically don't touch the files from the Pods project. In fact, you can completely ignore it. When you build the app, Xcode automatically compiles the Pods project and makes it part of your app. If you ever need to add a new library, you edit the Podfile, type `pod install` to update the Pods project, and re-open the workspace. Easy as pie.

➤ Build the app to make sure everything still works.

This seems like a good time to commit your first change in this new branch.

Note: Xcode gave me an error message when I tried to commit. It was confused about certain files, the ones with a **?** next to their names in the Project navigator. Apparently Xcode is not smart enough to recognize that these files were newly added to the project.

If this happens to you as well, you can fix this by right-clicking the filename and then choosing **Source Control** → **Add** from the popup menu. Unfortunately, you need to repeat that for about 90 files. Yikes!

Instead, go back to your Terminal window and type:

```
git add .
```

(The dot is important!) That accomplishes the same thing and is a lot less work. Xcode might not immediately pick up on these changes but there is nothing that restarting Xcode won't fix. Now all the **?**s will turn into **A**s, which means that these new files are now under source control and Xcode has been unconfused.

➤ Choose **Source Control → Commit** and make sure all the files are checked. The easiest way to do that is to right-click and choose **Check All**.

I used the commit message: "Added the AFNetworking library." Remember that this change is added only to the AFNetworking branch, not to the master branch.

> **AF? ASI? SV? MB?**
>
> As you may have noticed, UIKit classes start with the prefix UI, Core Graphics functions start with the prefix CG, Core Location classes with the prefix CL, Foundation stuff starts with NS, and so on.
>
> Unlike many other languages, Objective-C does not have the concept of namespaces. If UIKit were to name one of its classes Button, then any apps trying to use UIKit cannot have a class that is named Button also, because you are not allowed to have two different classes with the same name. Therefore, a two or three letter prefix is used to show to which library a class belongs.
>
> UIKit has claimed the prefix "UI" and therefore any class whose name starts with UI – such as `UIButton` – belongs to UIKit. If your name is Mickey Mouse and you want to create your own button and stick it into a library, then you should probably name it MMButton. That way you will avoid any naming conflicts with other libraries.
>
> This is only a convention and there is no one enforcing it but if you ever decide to write a reusable code library, pick a good prefix and put it in front of your class names.

Putting AFNetworking into action

➤ Remove the `parseJSON:` and `performStoreRequestWithURL:` methods from **SearchViewController.m**. Yup, that's right, you won't be needing them anymore.

The cool thing about AFNetworking is that it can perform the HTTP request on the server and automatically parses the returned data from JSON gobbledygook into an `NSDictionary`. You no longer need to parse the JSON data yourself.

➤ Add an import at the top of **SearchViewController.m**:

```
#import <AFNetworking/AFNetworking.h>
```

Note that this import uses < > brackets instead of the double quotes that you've been using so far. That's because AFNetworking is not part of the main StoreSearch project but the separate Pods project. (Recall that you also use < > brackets for importing system frameworks.)

➤ Change `searchBarSearchButtonClicked:` to the following:

```
- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
  if ([searchBar.text length] > 0) {
    [searchBar resignFirstResponder];

    _isLoading = YES;
    [self.tableView reloadData];

    _searchResults = [NSMutableArray arrayWithCapacity:10];

    NSURL *url = [self urlWithSearchText:searchBar.text];
    NSURLRequest *request = [NSURLRequest requestWithURL:url];

    AFHTTPRequestOperation *operation =
       [[AFHTTPRequestOperation alloc] initWithRequest:request];

    operation.responseSerializer =
                          [AFJSONResponseSerializer serializer];

    [operation setCompletionBlockWithSuccess:^
        (AFHTTPRequestOperation *operation, id responseObject) {

      NSLog(@"Success! %@", responseObject);

    } failure:^(AFHTTPRequestOperation *operation,
                                              NSError *error) {
      NSLog(@"Failure! %@", error);
    }];

    [_queue addOperation:operation];
  }
}
```

After you've created the `NSURL` object like before, you now put it into an `NSURLRequest` object. You use that request object to create a new `AFHTTPRequestOperation` object. As you can see, `AFHTTPRequestOperation` takes two blocks, one for success and one for failure.

The code in the success block is executed when everything goes right, while the code from the failure block gets executed if there is some problem making the request or when the response isn't valid JSON. For now you simply use an `NSLog()` to show which one of these blocks gets called.

The last thing the method does is this:

```
  [_queue addOperation:operation];
```

Not to make things too confusing, but this is a different type of queue than the GCD (Grand Central Dispatch) queues you've used before. They are similar in operation and function, hence the resemblance in name, but there are also some differences. In this case you're dealing with an `NSOperationQueue` object, and you'll need to make an instance variable for it.

➤ Add an instance variable for the `NSOperationQueue` object:

```
@implementation SearchViewController
{
  NSMutableArray *_searchResults;
  BOOL _isLoading;
  NSOperationQueue *_queue;
}
```

You will create this `_queue` object when the view controller gets instantiated, in other words in its init method.

➤ The Xcode template already provided you with a basic init method. Change that method to the following:

```
- (id)initWithNibName:(NSString *)nibNameOrNil
               bundle:(NSBundle *)nibBundleOrNil
{
  self = [super initWithNibName:nibNameOrNil
                         bundle:nibBundleOrNil];
  if (self) {
    _queue = [[NSOperationQueue alloc] init];
  }
  return self;
}
```

In previous tutorials you used `initWithCoder:` but here the view controller is not loaded from a storyboard or nib (only its view is). Look inside AppDelegate.m if you don't believe me. There you'll see the line that calls `initWithNibName:bundle:` to create and initialize the `SearchViewController` object. So that is the proper init method to add this code to.

With those changes made, you can run the app and see what AFNetworking makes of it.

➤ Run the app and search for something. After a second or two you should see the debug output say "Success!" followed by a dump of the JSON dictionaries. Excellent!

The app is not doing anything yet with the search results, but you already wrote all the code you need for that, so let's put it in the success and failure blocks.

➤ Change the success block to:

```
  [self parseDictionary:responseObject];
  [_searchResults sortUsingSelector:@selector(compareName:)];

  _isLoading = NO;
  [self.tableView reloadData];
```

(In case you're not entirely sure where to find the success block: you put this in the place of the `NSLog(@"Success! %@" ...)` line.)

This takes the object from the `responseObject` parameter, which is actually an `NSDictionary`, and calls `parseDictionary:` to turn its contents into `SearchResult` objects, just like you did before. Then you sort the results and put everything into the table.

▸ Change the failure block to:

```
  [self showNetworkError];

  _isLoading = NO;
  [self.tableView reloadData];
```

(You put this in the place of the `NSLog(@"Failure! %@" ...)` line.)

Here you simply call the `showNetworkError` message to tell the user that something went wrong. Note that you have to put `[self.tableView reloadData]` in both blocks, because in both cases you will need to refresh the contents of the table view!

▸ Run the app. The search should work again. You have successfully replaced the old networking code with AFNetworking.

Tip: Whenever you use an API that can execute blocks after some operation completes, you have to check the documentation to make sure whether this block is performed on the main thread or not. In the case of `AFHTTPRequestOperation` they are (I checked!), so you can safely call UIKit code here. If the block got executed on another thread, then you would have to wrap this in a new `dispatch_async()` block instead. Fortunately AFNetworking makes it easy and you don't have to worry about any of that.

▸ This is a good time to commit your changes.

Canceling operations

What happens when a search takes very long and the user already starts a second search when the first one is still going? The app doesn't disable the search bar so it's possible for the user to pull this off. When dealing with networking – or any asynchronous process, really – you have to think these kinds of situations through.

There is no way to predict what happens, but it will most likely be a strange experience for the user. He might see the results from his first search, which he is

no longer expecting (confusing!), only to be replaced by the results of the second search a few seconds later. But there is no guarantee the first search completes before the second, so the results from search 2 may arrive first and then get overwritten by the results from search 1, which is definitely not what the user wanted to see either.

Because you're no longer blocking the main thread, the UI always accepts user input, and you cannot assume the user will sit still and wait until the request is done. You can usually fix this dilemma in one of two ways:

1. Disable all controls. The user cannot tap anything while the operation is taking place. This does not mean you're blocking the main thread; you're just making sure the user cannot mess up the order of things.
2. Cancel the on-going request when the user starts a new one.

For this app you're going to pick the second solution. Every time the user performs a new search you cancel the previous request. Since you're using an `NSOperationQueue`, you can simply tell the queue to cancel any operations that are still in the queue.

➤ Make the following change to `searchBarSearchButtonClicked:`

```
- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
  if ([searchBar.text length] > 0) {
    [searchBar resignFirstResponder];

    [_queue cancelAllOperations];

    . . .
```

With `cancelAllOperations` you make sure that no old searches can ever get in the way of the new search.

➤ Canceling an `AFHTTPRequestOperation` invokes its failure block, so add these lines to the failure block to prevent the app from showing an error message:

```
} failure:^(AFHTTPRequestOperation *operation, NSError *error) {

  if (operation.isCancelled) {
    return;
  }

  . . .
```

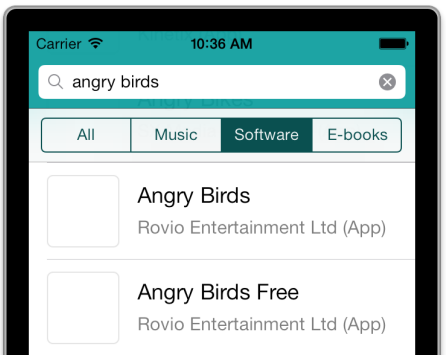
➤ Test the app with and without this call to `cancelAllOperations` to experience the difference. If you use the SpeedLimit preferences pane you can delay each query by a few seconds so it's easier to get two requests running at the same time.

➤ If you're satisfied it works, commit the changes to the repository.

> **Note:** Maybe you don't think it's worth making a commit when you've only changed a few lines, but many small commits are often better than a few big ones. Each time you fix a bug or add a new feature is a good time to commit.

Improving the search results

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 you pick the category that you want to search in. It looks like this:



Searching in the Software category

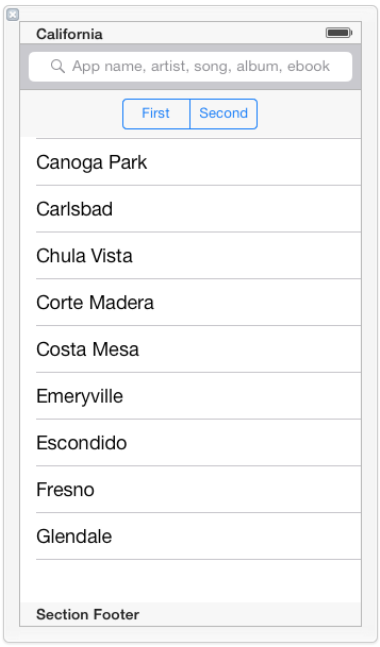
This type of control is called a **segmented control** and is used to pick one option out of multiple choices.

➤ Open **SearchViewController.xib** in Interface Builder.

➤ 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.

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

The design now looks like this:



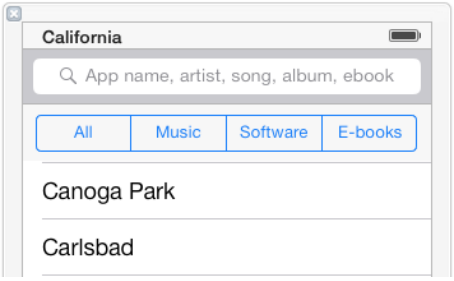
The Segmented Control sits in a Navigation Bar below the Search Bar

› Select the Segmented Control. Set its **Width** to 300 points (make sure you change the width of the entire control, not of the individual segments).

› In the **Attributes inspector**, set the number of segments to 4.

› Change the title of the first segment to **All**. Then select the second segment and set its title to **Music**. The title for the third segment should be **Software** and the fourth segment is **E-Books**.

The nib should look like this now:



The finished Segmented Control

› Add a new outlet property for the Segmented Control to **SearchViewController.m**:

```
@property (nonatomic, weak) IBOutlet UISegmentedControl
                                          *segmentedControl;
```

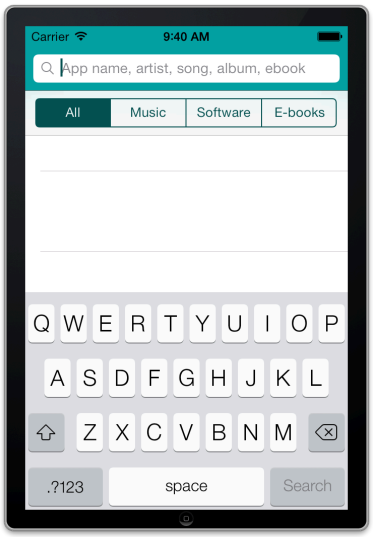
➤ Also add a very basic version of the action method for the Segmented Control to the bottom of the source file:

```
- (IBAction)segmentChanged:(UISegmentedControl *)sender
{
  NSLog(@"segment changed: %d", sender.selectedSegmentIndex);
}
```

➤ Go back to Interface Builder and **Ctrl-drag** from File's Owner to the Segmented Control to connect the **segmentedControl** outlet.

➤ **Ctrl-drag** from the Segmented Control back to File's Owner and connect it to the **segmentChanged:** action method. Remember that connecting actions is the other way around from connecting outlets.

➤ Run the app to make sure everything still works. Tapping a segment should log a number (the index of that segment) to the debug pane.



The segmented control in action

Notice that the first row of the table view is partially obscured again. Because you placed a navigation bar below the search bar, you need to add another 44 points to the table view's content inset.

➤ Change that line in `viewDidLoad` to:

```
self.tableView.contentInset = UIEdgeInsetsMake(108, 0, 0, 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 perform the same search for that new product category.

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

➤ Rename the `searchBarSearchButtonClicked:` method to `performSearch` (without a colon at the end).

You're putting the search logic into a separate method that can be invoked from more than one place. Xcode will now complain that there is no such thing as "searchBar". Previously that was the parameter of the `searchBarSearchButtonClicked:` method but this new `performSearch` method has no parameter.

➤ Simply replace the occurrences of `searchBar` in this method with `self.searchBar`.

➤ Now add a new version of the `searchBarSearchButtonClicked:` back into the source code, above `performSearch`:

```
- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar
{
  [self performSearch];
}
```

➤ Replace the `segmentChanged:` method with:

```
- (IBAction)segmentChanged:(UISegmentedControl *)sender
{
  if (_searchResults != nil) {
    [self performSearch];
  }
}
```

The app will always call `performSearch` if the user presses the Search button on the keyboard, but in the case of tapping on the Segmented Control it will only do a new search if the user has already performed a search before.

➤ Run the app and verify that searching still works. When you tap on the different segments the search should be performed again as well.

Note: The second time you search for the same thing the app may return results very quickly. The networking layer is now returning a *cached* response so it doesn't have to download the whole thing again, which is usually a performance gain on mobile devices. (There is an API to turn off this caching behavior if that makes sense for your app.)

You still have to tell the app to use the category from the selected segment for the search. You've already seen that you can get the index of the selected segment with the `selectedSegmentIndex` property. This returns an `NSInteger` value (0, 1, 2, or 3).

➤ Change the `urlWithSearchText:` method so that it accepts this `NSInteger` as a parameter and then builds up the request URL accordingly:

```
- (NSURL *)urlWithSearchText:(NSString *)searchText
                    category:(NSInteger)category
{
  NSString *categoryName;
  switch (category) {
    case 0: categoryName = @""; break;
    case 1: categoryName = @"musicTrack"; break;
    case 2: categoryName = @"software"; break;
    case 3: categoryName = @"ebook"; break;
  }

  NSString *escapedSearchText = [searchText
           stringByAddingPercentEscapesUsingEncoding:
                                        NSUTF8StringEncoding];

  NSString *urlString = [NSString stringWithFormat:
  @"http://itunes.apple.com/search?term=%@&limit=200&entity=%@",
                                escapedSearchText, categoryName];

  NSURL *url = [NSURL URLWithString:urlString];
  return url;
}
```

This first turns the category index from a number into a string. (Note that the category index is passed to the method as a new parameter.)

Then it puts this string behind the `&entity=` parameter in the URL. For the "All" category, the entity value is empty but for the other categories it is "musicTrack", "software" and "ebook", respectively.

➤ In the `performSearch` method, change the line that used to call `urlWithSearchText:` into the following:

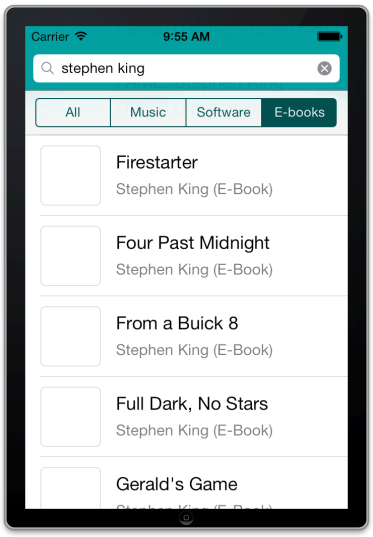
```
NSURL *url = [self urlWithSearchText:self.searchBar.text
           category:self.segmentedControl.selectedSegmentIndex];
```

And that should do it.

Note: You could have used `self.segmentedControl.selectedSegmentIndex` directly inside `urlWithSearchText:` instead of passing the category index as a parameter. Using the parameter is the better design, though. It makes it possible to reuse the same method with a different type of control, should you decide that a Segmented Control isn't really the right component for this app.

> It is always a good idea to make methods as independent from each other as possible.

➤ Run the app and search for "stephen king". In the All category that gives results for anything from movies to podcasts to audio books. In the Music category it matches mostly artists with the word "King" in their name. There doesn't seem to be a lot of Stephen King-related software, but in the E-Books category you finally find some of his novels.

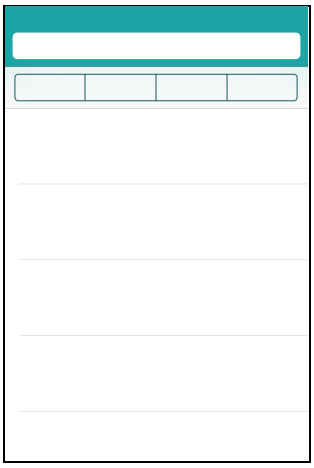


You can now limit the search to just e-books

This finalizes the UI design of the main screen, so now you can make the launch images and add them to the project. I already prepared these images so all you have to do is add them.

➤ Drag the launch images from this chapter's resources into the asset catalog.

Notice that I removed all the text from this initial image. That's important because later in this tutorial you will be translating the app to another language and then the on-screen texts will be different.



The launch image

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

Downloading the artwork images

The JSON search results contain a number of URLs to images and you put two of those – `artworkURL60` and `artworkURL100` – into the `SearchResult` object. Now you are going to download these images over the internet and put them into the table view cells.

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

http://a1.mzstatic.com/us/r1000/059/Music/55/f4/44/mzi.xaimsnky.100x100-75.jpg

If you click on that link, it will open that picture in a new web browser window. The server where this picture is stored is apparently not itunes.apple.com but a1.mzstatic.com, but that doesn't matter anything to the app. As long as it has a valid URL, it will just go fetch the file at that location, no matter where it is and no matter what kind of file that is.

There are various ways that you can download files from the internet. You're going to use AFNetworking because it has a very cool `UIImageView` category for this. Of course, you'll be downloading these images asynchronously!

First, you will move the logic for configuring the contents of the table view cells into the `SearchResultCell` class. That's a better place for it. Logic related to an object should live inside that object as much as possible, not somewhere else. Many developers have a tendency to stuff everything into their view controllers, but if you can move some of the logic into other objects that makes for a much cleaner program.

› Add the following method signature to **SearchResultCell.h**:

```
- (void)configureForSearchResult:(SearchResult *)searchResult;
```

That also requires a `@class` statement to let the compiler know what sort of thing `SearchResult` is.

➤ Add the following line above the `@interface` declaration:

```
@class SearchResult;
```

➤ Now import the `SearchResult` class in **SearchResultCell.m**:

```
#import "SearchResult.h"
```

➤ And implement the `configureForSearchResult:` method:

```
- (void)configureForSearchResult:(SearchResult *)searchResult
{
  self.nameLabel.text = searchResult.name;

  NSString *artistName = searchResult.artistName;
  if (artistName == nil) {
    artistName = @"Unknown";
  }

  NSString *kind = [self kindForDisplay:searchResult.kind];
  self.artistNameLabel.text = [NSString stringWithFormat:
                                  @"%@ (%@)", artistName, kind];
}
```

This is really the same as what you used to do in `cellForRowAtIndexPath`. The only problem is that this class doesn't have the `kindForDisplay:` method.

➤ Cut the `kindForDisplay:` method out of **SearchViewController.m** and paste it below `configureForSearchResult:` in **SearchResultCell.m**.

It's easy to move this method from one class to another because it doesn't depend on any instance variables. It is completely self-contained. You should strive to write your methods in that fashion as much as possible.

➤ Finally, change `cellForRowAtIndexPath` to the following:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  if (_isLoading) {
    . . .
  } else if ([_searchResults count] == 0) {
    . . .
```

```
  } else {
    SearchResultCell *cell = (SearchResultCell *)[tableView
         dequeueReusableCellWithIdentifier:
             SearchResultCellIdentifier forIndexPath:indexPath];

    SearchResult *searchResult = _searchResults[indexPath.row];
    [cell configureForSearchResult:searchResult];

    return cell;
  }
}
```

You just moved some code from one class (`SearchViewController`) into another (`SearchResultCell`). This was a small refactoring that is 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!

▸ Run the app to make sure everything still works as before.

OK, here comes the cool part. You will now use AFNetworking's `UIImageView` category to load the image and automatically put it into the table view's image view.

Remember that a category can be used to extend the functionality of an existing class and that's exactly what AFNetworking's developers did. Because loading images and showing them in `UIImageViews` is a very common thing, they provided this really convenient category that lets you pull this off with just one line of code.

▸ First, import AFNetworking at the top of **SearchResultCell.m**:

```
#import <AFNetworking/UIImageView+AFNetworking.h>
```

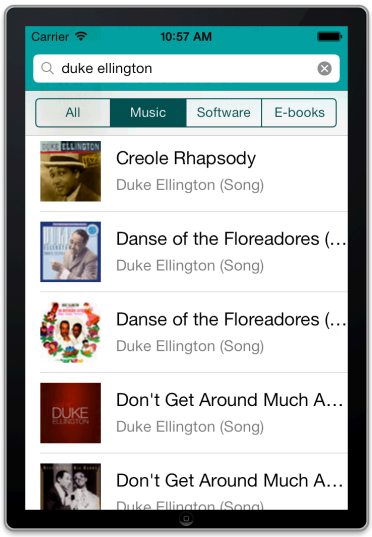
You don't need to import all of AFNetworking, only the category with the name **UIImageView+AFNetworking**.

▸ Add the following line to the bottom of `configureForSearchResult:`

```
- (void)configureForSearchResult:(SearchResult *)searchResult
{
  . . .

  [self.artworkImageView setImageWithURL:
                [NSURL URLWithString:searchResult.artworkURL60]
        placeholderImage:[UIImage imageNamed:@"Placeholder"]];
}
```

This tells AFNetworking to load the image from `artworkURL60` and to place it in the cell's image view. While the image is loading the image view displays the placeholder image that you added to the asset catalog earlier.

▸ Run the app and look at those icons!



The app now downloads the album artwork

How was that for easy? You're not quite done yet. Remember that table view cells can be reused, so it's theoretically possible that you're scrolling through the table and some cell is about to be reused while its previous image is still loading. You no longer need that image so you should really cancel the pending download. Table view cells have a special method named `prepareForReuse` that is ideal for this.

▸ Add the following method to **SearchResultCell.m**:

```
- (void)prepareForReuse
{
  [super prepareForReuse];
  [self.artworkImageView cancelImageRequestOperation];
  self.nameLabel.text = nil;
  self.artistNameLabel.text = nil;
}
```

Here you cancel any image download that is still in progress and for good measure you also clear out the text from the labels. It's always a good idea to play nice.

Exercise. Put an `NSLog()` in the `prepareForReuse` method and see if you can trigger it. □

On a decent Wi-Fi connection, loading these images is very fast. You almost cannot see that it happens, even if you scroll quickly. It probably helps that the image files

are small (only 60 by 60 pixels) and that the iTunes servers are fast. That is key to having a snappy app: don't download more data than you need to.

Caching

Depending on what you searched for, you may have noticed that many of the images were the same. For example, my search for Duke Ellington's music had many identical album covers in the search results. AFNetworking is smart enough not to download identical images – or at least images with identical URLs – twice. That principle is called **caching** and it's very important on mobile devices.

Mobile developers are always trying to optimize their apps to do as little as possible. If you can download something once and then use it over and over, that's a lot more efficient than re-downloading it all the time.

There is more than just images that you can cache. You can also cache the results of big computations, for example. Or views, as you have been doing in the previous chapters. 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.

Some caches are in-memory only where the cached data stays in the computer's working memory, but it is also possible to cache the data in files on the disk. You don't do that for this app because the images are not that important, but saving data to cache files is a common thing for apps to do. Your app even has a special directory for it, named Library/Caches.

AFNetworking uses its own image cache to store the images that are downloaded by the `UIImageView` category. This is completely automatic, so that takes another burden off your shoulders. Using great third-party libraries can make your developer life a lot easier!

Merging the branch

This concludes the section on talking to the web service and downloading images. Later on you'll tweak the web service requests a bit more (to include the user's language and country) but for now you're done with this feature. I hope you got a good glimpse of what is possible with web services and how easy it is to build this into your apps with a third-party library such as AFNetworking.

- Commit these latest changes to the repository.

Now that you've completed a feature, you can merge this temporary branch back into the master branch. To do that, you first have to return to the master branch. This is possible to do in Xcode but only if you like pain. I gave up after several tries because Xcode kept messing up my files. So you're in for some command line fun.

➤ 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. Even though you just committed your files, Xcode may have made changes to some of the project files after that commit. These files are part of the Git repository too, but the changes are inconsequential and not worth committing.

```
git checkout master
```

This switches the current branch back to the master branch.

```
git merge AFNetworking
```

This merges the changes from the AFNetworking branch back into the master branch. If you get an error message at this point, then simply do `git stash` again and repeat the 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. (It may take a minute for Xcode to realize it is now on the master branch, so don't freak out if it still says "AFNetworking" in the Source Control menu.)

Git is a pretty awesome tool but it takes a while to get familiar with. Unfortunately, Xcode's support for things like merges is very spotty and you're better off using the command line for the more advanced commands. It's well worth learning!

You can find the project files for the app up to this point under **05 - AFNetworking** in the tutorial's Source Code folder.

Manual Memory Management

Before ARC (Automatic Reference Counting) you had to do something called "manual memory management" and it was just a hassle. Thanks to advances in Objective-C you no longer have to do any of that. The compiler will figure

out automatically how to do this memory management and takes care of it behind the scenes.

Any developer who's been around the block will tell you: ARC is great. However, a lot of third-party libraries were written before ARC was available and they still do manual memory management. When you use third-party libraries in your apps (and you most likely will) then you may still be confronted with it. So it's good to know a bit about manual memory management and how to convert it to ARC.

If you pick up iPhone programming books that were written before iOS 5 came out (October 2011) or read through older blog posts, then you'll often see source code examples that still use manual memory management. You should know how to interpret this code if you want to use it in your own apps.

If you have a source code file that uses manual memory management, you can simply add it to your project and disable ARC just for that file with a compiler flag, **-fno-objc-arc** (you can specify this in the Project Settings screen, under Build Phases, Compile Sources). However, if the source file is short or if you just have a small code snippet that you want to use, you can also remove all the manual memory management code to make it work under ARC.

So what exactly is this manual memory management all about? Remember that with ARC you have strong and weak pointers. As long as there is at least one strong pointer to an object, that object stays alive. As soon as there are no more strong pointers, the object is deallocated. With manual memory management there are no strong or weak pointers, just regular plain old pointers, and you have to explicitly say that you want to keep an object alive (or not).

You do this with three methods: `retain`, `release`, and `autorelease`. To keep an object alive, you call `retain`, like this:

```
- (void)eatIceCream:(IceCream *)iceCream
{
  _myIceCream = [iceCream retain];
  [_myIceCream eat];
}
```

This assigns the object from the `iceCream` parameter to an instance variable named `_myIceCream`. This example assumes you want to keep the object alive beyond this method, so you must `retain` it. By calling `retain`, you're saying that you want `_myIceCream` to act as a strong pointer.

If you no longer need an object, you have to `release` it. After the last owner has released an object, it will be deallocated. That is similar to there being no more strong pointers under ARC. When you do `[_myIceCream release]`, you're

saying that you no longer want `_myIceCream` to be a strong pointer to this object.

```
- (void)dealloc
{
  [_myIceCream release];
  [super dealloc];
}
```

Under manual memory management, you have to `release` all your instance variables (and retained properties) in your `dealloc` method, otherwise these objects will stay in memory forever and the app will have the infamous memory leak. You also need to call `[super dealloc]`, something that is not allowed under ARC.

The third memory management method is `autorelease`. That also releases the object in question but not immediately. It puts this method into the **autorelease pool**, which is drained after the current event has been completely handled.

So far so good. You call `retain` to keep an object alive (make a strong pointer) and `release` or `autorelease` to give up the strong pointer. However, there are some special rules and situations that you need to be aware of or things will go horribly wrong.

Under certain circumstances, `retain` is done for you. When you call a method whose name starts with `alloc`, `init`, `new` or `copy`, then that method already returns a retained object. You don't need to call `retain` on it again. For example,

```
NSString *s = [[NSString alloc] initWithFormat:
                                      @"Hello, %@", yourName];
```

will return a new `NSString` object that is retained. You are responsible for releasing it when you're done with the string. However, the following,

```
NSString *s = [NSString stringWithFormat:
                                  @"Hello, %@", yourName];
```

returns an `NSString` object that is autoreleased. Because the name of the method does not begin with `alloc`, `init`, `new` or `copy`, the string object is not retained for you. This object will remain valid only until the end of your method (unless you `retain` it on purpose) and you may not `release` it again.

If this sounds like a big nightmare, then you're right. It's really easy to make a small mistake. If you forget a `retain` at some point, then your app may crash because it attempts to use an object that no longer exists. You didn't `retain` it so you didn't become an owner of the object and the other owners may have all released it so there is no one keeping it alive.

On the other hand, if you forget a `release` then the object will stay in memory forever and you'll have a so-called **memory leak**. With enough memory leaks, your app will run out of working space and it will crash as well.

With ARC you no longer have to worry about retains and releases. If you want to keep an object alive, simply make a strong pointer. If you no longer need the object, set the pointer to `nil`. That's much easier!

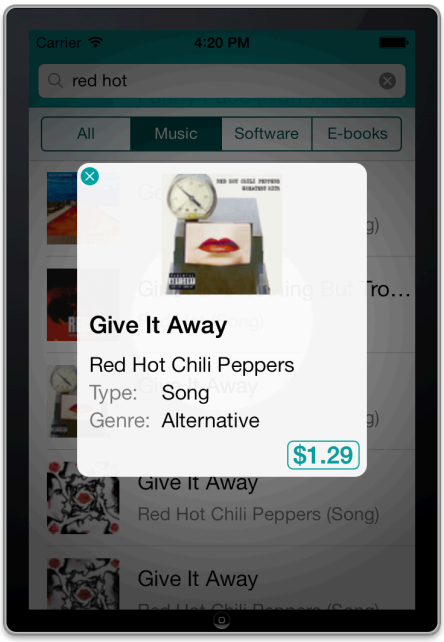
However, in the case that you need to convert code with `retain`s and `release`s to ARC, often you can simply remove any line that says `retain`, `release` or `autorelease`. Xcode also has an automated conversion tool that you can find under Edit → Refactor → Convert to Objective-C ARC.

If you want to learn more about the differences between manual memory management and ARC, and how to convert old code to work under ARC, then check out these tutorials I wrote:
http://www.raywenderlich.com/5677/beginning-arc-in-ios-5-part-1
http://www.raywenderlich.com/5773/beginning-arc-in-ios-5-tutorial-part-2

To learn about the manual memory management rules, read the following document from Apple. Following these rules is very important when you're dealing with manually memory-managed code.
http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmRules.html.

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

As you can see, 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 the *view controller containment* API. You will also learn to draw your own gradients with Core Graphics and to make cool keyframe animations with Core Animation. Fun times ahead!

The to-do list for this section is:

- Design the Detail screen in a new nib.
- Show this screen when the user taps on a row in the table.
- Put the data from the `SearchResult` into the screen. This includes the item's price, formatted in the proper currency.
- Make the Detail screen appear with a cool animation.

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

The Detail View Controller

➤ Add a new file to the project. Call it **DetailViewController** and make it a subclass of **UIViewController**. Enable the **With XIB for user interface** option.

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

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

➤ Drag a new **View** into the nib. Using the **Size inspector**, make it 240 points wide and 260 high. Center the view in the window.

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

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

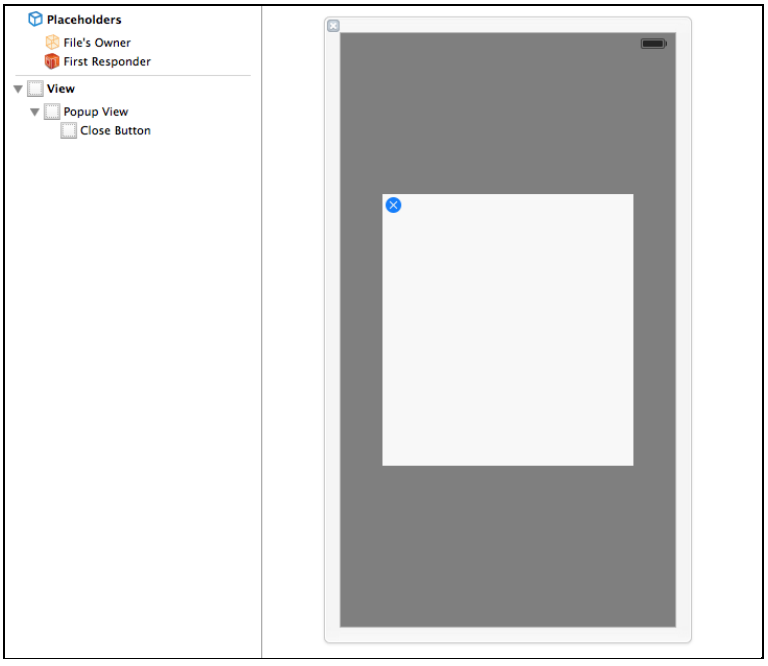
➤ Drag a **Button** into the nib. Place it somewhere on the Popup View. In the **Attributes inspector**, change **Image** to **CloseButton** (you already added this image to the asset catalog earlier).

➤ 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 Popup View (at X = 3 and Y = 0).

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

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

The design should look as follows:



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

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

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

```
- (IBAction)close:(id)sender
{
  [self dismissViewControllerAnimated:YES completion:nil];
}
```

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

➤ Connect this action method to the **X** button's Touch Up Inside event in Interface Builder. (As before, Ctrl-drag from the button to File's Owner.)

> **Note**: File's Owner for this nib is `DetailViewController`. When you edited SearchViewController.xib, File's Owner was the `SearchViewController`. Now that you're editing the nib for the Detail screen, the owner of this file is the `DetailViewController`. Therefore when you drag from the close button to File's Owner you will only see the actions that are declared in `DetailViewController`, not those in `SearchViewController`.

➤ Open **SearchViewController.m** and add an import:

```
#import "DetailViewController.h"
```

➤ Change `didSelectRowAtIndexPath` to the following:

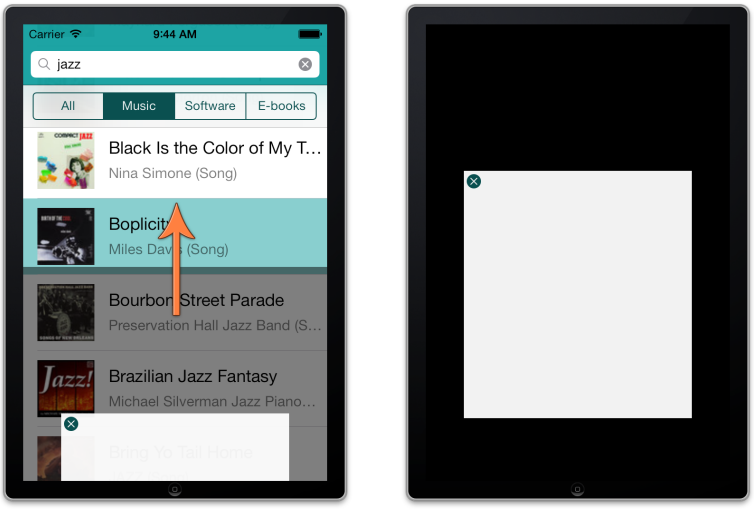
```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  [tableView deselectRowAtIndexPath:indexPath animated:YES];

  DetailViewController *controller =
      [[DetailViewController alloc] initWithNibName:
                            @"DetailViewController" bundle:nil];

  [self presentViewController:controller animated:YES
                   completion:nil];
}
```

Because this app uses nibs and not storyboards, you cannot make segues by drawing an array between two different view controllers. To show a new view controller you have to alloc and init it yourself and then present it. This is the equivalent of making a modal segue.

Let's see how well this works.

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

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



What happens when you present the Detail screen modally

(In addition, if you run the app on the 3.5-inch Simulator, as in the above screenshots, the popup isn't neatly centered.)

Clearly, presenting this new screen as a modal view controller isn't going to achieve the effect we're after.

There are two possible solutions:

1. Don't have a `DetailViewController`. You can just load the new view from the 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 and 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`.

Before iOS 5 it was a bit tricky to put more than one view controller on the same screen. The motto used to be: one screen, one view controller. However, on devices with larger screens such as the iPad that became inconvenient –you often want one area to be controlled by one view controller and another area by another view controller – so now view controllers are allowed to be part of other view controllers if you follow a few rules.

These "view controller container APIs" are not limited to just the iPad; you can take advantage of them on the iPhone as well. So as of iOS 5 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.

Obviously, you're going to go for this option because it allows you to keep all the logic for the Detail screen in the `DetailViewController`.

Using the view controller containment APIs

➤ Change `didSelectRowAtIndexPath` to the following:

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  [tableView deselectRowAtIndexPath:indexPath animated:YES];

  DetailViewController *controller = [[DetailViewController
     alloc] initWithNibName:@"DetailViewController" bundle:nil];

  [self.view addSubview:controller.view];
  [self addChildViewController:controller];
  [controller didMoveToParentViewController:self];
}
```

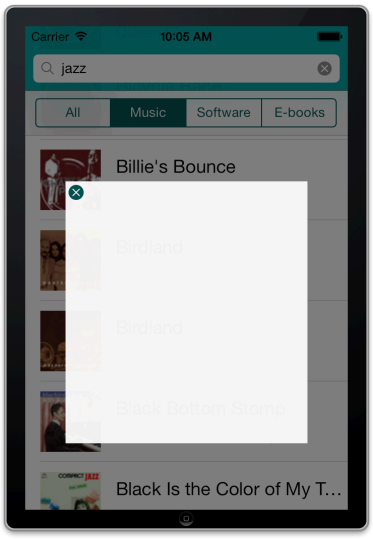
You no longer use `presentViewController:animated:completion:` to show the new `DetailViewController` as a modal screen, but you add it as a child view controller in three steps.

These are the minimum required steps to add the contents of one view controller to another:

1. First, add the new view controller's view as a subview. This places it on top of the table view, search bar and segmented control.
2. Then tell the `SearchViewController` that the `DetailViewController` 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, as I shall demonstrate in a short while.
3. Tell the new view controller that it now has a parent view controller with `didMoveToParentViewController:`.

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

➤ Run the app and tap a row. The pop-up will now appear on top of everything else.



The Detail pop-up is now transparent

Because you made the background color of the main view in **DetailViewController.xib** 50% transparent black, you can still see the table view and the other controls in the background but tapping on them has no effect. Because the new view covers the whole screen (even though parts of it are see-through), you cannot tap on underlying items.

Tapping the close button also no longer works. You need to change the `close:` action method because that still tries to dismiss the modal view controller the old way.

▸ In **DetailViewController.m**, change the `close:` method to:

```
- (IBAction)close:(id)sender
{
  [self willMoveToParentViewController:nil];
  [self.view removeFromSuperview];
  [self removeFromParentViewController];
}
```

This is essentially the inverse of what you did to embed the view controller. First you call `willMoveToParentViewController:` 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 you call `removeFromParentViewController` to truly dispose of the view controller.

▸ Run the app. Now the close button should work.

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

▸ Add a `dealloc` method to **DetailViewController.m**:

```
- (void)dealloc
{
  NSLog(@"dealloc %@", self);
}
```

▸ Run the app and verify that the `dealloc` method is indeed being called after you press the close button.

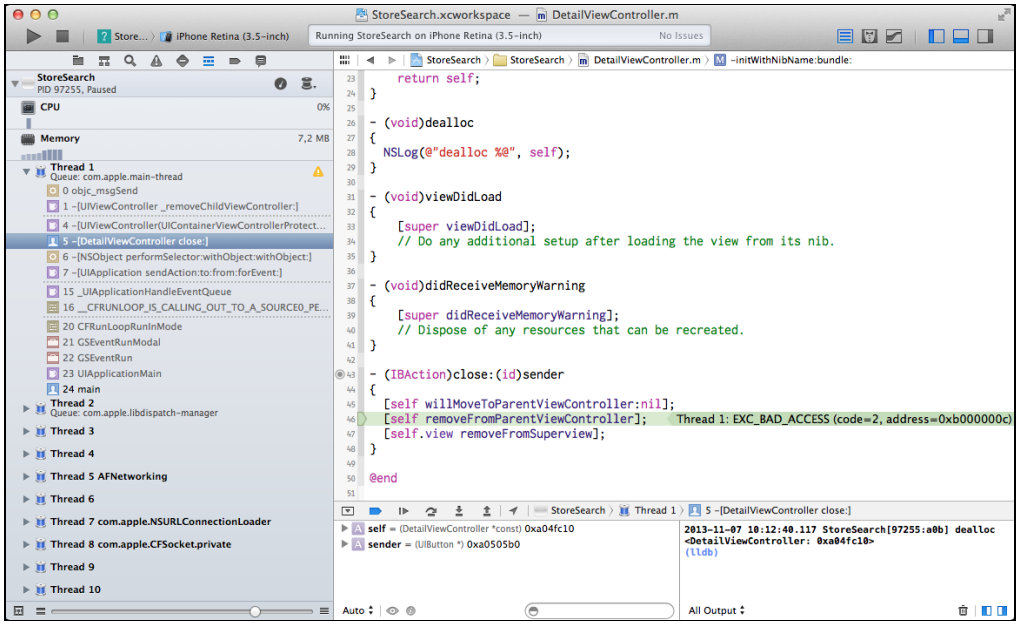
The dreaded EXC_BAD_ACCESS crash

It's important that the steps in the `close:` method occur in the right order. For fun, change it to the following:

```
- (IBAction)close:(id)sender
{
  [self willMoveToParentViewController:nil];
  [self removeFromParentViewController];
  [self.view removeFromSuperview];
}
```

The order of the `removeFromParentViewController` and `removeFromSuperview` calls has now changed. Run the app, tap a row, and press the close button on the pop-up.

What happens next may vary from instant to instant, but most likely the app will crash and Xcode will end up in the debugger screen. (If not, press the X button again.)



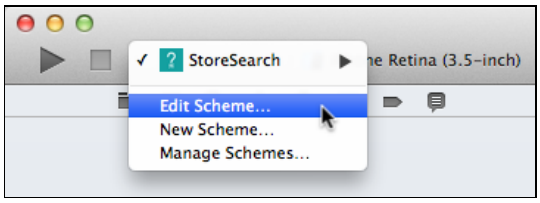
Crash!

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 ARC 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.

In this case the debugger already points at the line with the problem, but you may not always be so lucky (often it will point at a line in **main.m** which is not very helpful).

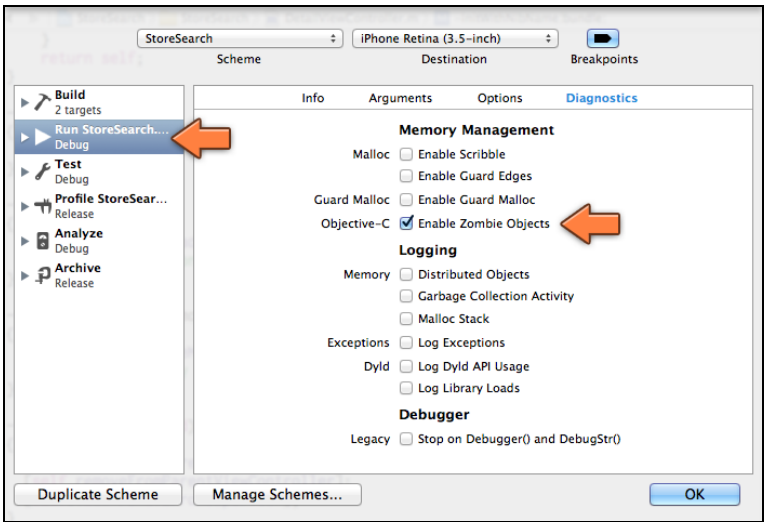
Fortunately, Xcode has a great tool that can help you debug these sorts of situations, called Zombie Objects (also known as the NSZombieEnabled setting). If you get an EXC_BAD_ACCESS error, then enable zombies and try again.

➤ To enable this debug tool, you have to edit the project's scheme. Go to the box that says **StoreSearch > iPhone Retina (3.5-inch)** in the top-left corner of the Xcode window, click on the left-hand side to open a menu and pick the Edit Scheme option:



Choosing the Edit Scheme option

This opens a new panel. Select **Run StoreSearch** on the left and then switch to the Diagnostics tab. Check the **Enable Zombie Objects** box and press OK.



Enabling the Zombie Objects tool

➤ Run the app and make it crash again.

The Debug pane now shows the following message:

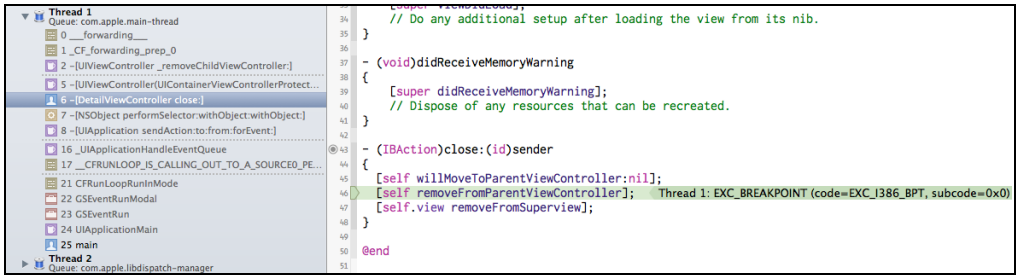
```
dealloc <DetailViewController: 0x68bc550>

*** -[DetailViewController _parentViewController]: message sent to
deallocated instance 0x68bc550
```

The first line is the `NSLog()` from the `dealloc` method of `DetailViewController`, but the next line is the interesting one. Rather than crashing the app without explanation, the Zombie tool has caught the error.

Apparently you're trying to send a message to a "deallocated instance". In other words, one of your objects got deallocated because it was no longer being used – there were no more strong references to it – yet you still tried to call a method on that object.

According to the error message, the object in question is `DetailViewController` and the method is `_parentViewController`. That is not any of your methods, so it must be something inside UIKit that went wrong. (But it's likely still something that you did!)

That's one part of the puzzle solved. You know the name of the object and the name of the method. Now you need to know where the crash happened. Fortunately, Xcode shows the exact spot:



The Xcode debugger shows where the app crashed

(If instead you see a bunch of assembly code, then look at the call stack on the left and click on the first line from the top that isn't grayed out. That is your source code; everything that is grayed out is from a system library.)

Often when the app crashes with an EXC_BAD_ACCESS error, the Xcode debugger isn't of much help. But with zombies enabled, the debugger can pinpoint the exact location that caused the problem. The crash appears to happen in the call to `[self removeFromParentViewController]` but it is not entirely clear yet why. Let's add some `NSLog()` statements to figure out exactly what's happening:

▸ Change the `close:` method to:

```
- (IBAction)close:(id)sender
{
```

```
    [self willMoveToParentViewController:nil];
    NSLog(@"Before removeFromParentViewController");
    [self removeFromParentViewController];
    NSLog(@"After removeFromParentViewController");
    [self.view removeFromSuperview];
}
```

▸ Run the app again and reproduce the crash. The Xcode Debug pane now says:

```
Before removeFromParentViewController

dealloc <DetailViewController: 0x689d330>

*** -[DetailViewController _parentViewController]: message sent to
deallocated instance 0x689d330
```

The line "After removeFromParentViewController" is never printed. So it looks like the `DetailViewController` object gets deallocated at some point during the execution of the `removeFromParentViewController` method, because the `NSLog()` from `dealloc` comes *after* the text "Before removeFromParentViewController".

It seems that `removeFromParentViewController` tries to send a message to the `DetailViewController` after it has been deallocated and that makes the app crash.

The interesting thing is that this doesn't happen if you change the order of the method calls back to what it was, so that `[self.view removeFromSuperview]` happens before the call to `removeFromParentViewController:`.

```
- (IBAction)close:(id)sender
{
  [self willMoveToParentViewController:nil];
  [self.view removeFromSuperview];
  NSLog(@"Before removeFromParentViewController");
  [self removeFromParentViewController];
  NSLog(@"After removeFromParentViewController");
}
```

Now the debug output is:

```
Before removeFromParentViewController

After removeFromParentViewController

dealloc <DetailViewController: 0x68c9350>
```

And the app doesn't crash anymore. As I said, using ARC these sorts of crashes should be rare but not impossible. Simply by putting two lines in the wrong order you managed to make the app crash on a memory management related error. Fortunately you have the Zombie Objects tool to help find the exact point of the crash.

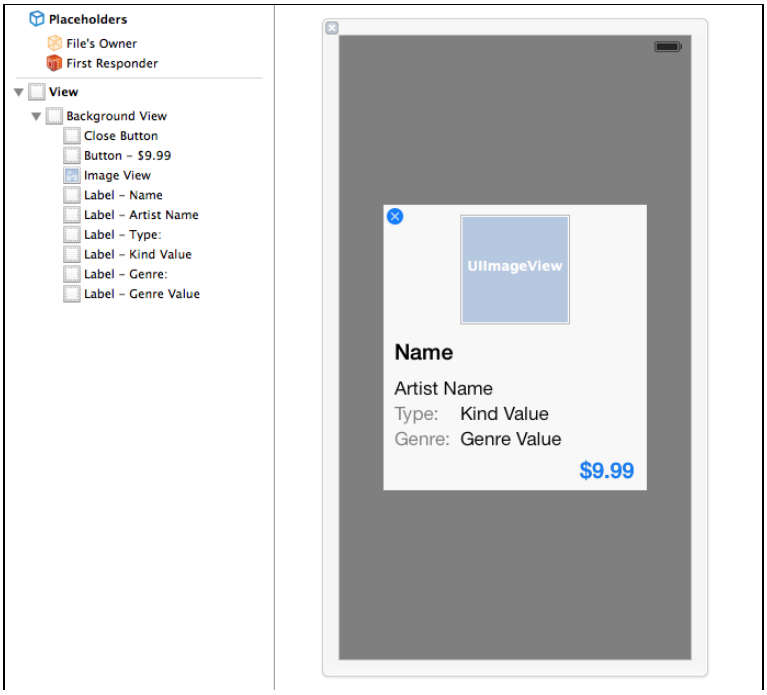
Note that you shouldn't enable Zombie Objects all the time. With zombies enabled, whenever an object gets deallocated its memory isn't actually returned to the free pool of memory. So it's possible that your app will run out of free memory if you keep Zombie Objects enabled all the time.

My advice: only enable Zombie Objects when you get an EXC_BAD_ACCESS crash and disable them once you've fixed the bug.

➤ Disable zombie objects and commit your changes. Oh, you can remove the `NSLog()` statements as well.

Adding the rest of the controls

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



The Detail screen with the rest of the controls

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

Some suggestions for the dimensions:

Control	X	Y	Width	Height
Image View	70	9	100	100
Name label	10	110	220	48
Artist Name label	10	158	220	19
Type: label	10	181	43	19
Kind Value label	70	181	160	19
Genre: label	10	204	52	19
Genre Value label	70	204	160	19
$9.99 button	174	230	60	24

› The **Name** label's **Lines** attribute is set to 0 because it may need to fit more than one line of text. Its font is **System Bold 20.0**. Set **Autoshrink** to **Minimum Font Size** 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 20.0**. In a moment you will give this button a background image but you won't do this from within Interface Builder.

› You shouldn't have to change the font for the other labels; they use the default value of System 17.0.

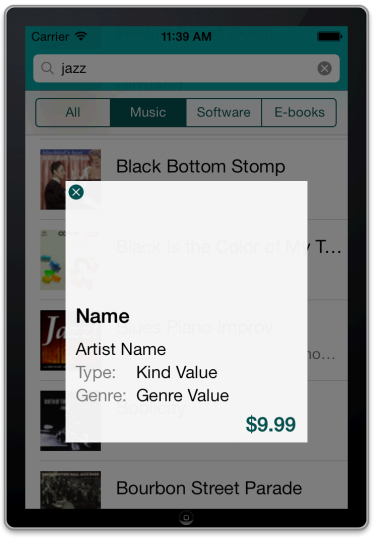
› Set the **Color** for the **Type:** and **Genre:** labels to 50% opaque black.

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

```
@interface DetailViewController ()
@property (nonatomic, weak) IBOutlet UIView *popupView;
@property (nonatomic, weak) IBOutlet UIImageView
                                        *artworkImageView;
@property (nonatomic, weak) IBOutlet UILabel *nameLabel;
@property (nonatomic, weak) IBOutlet UILabel *artistNameLabel;
@property (nonatomic, weak) IBOutlet UILabel *kindLabel;
@property (nonatomic, weak) IBOutlet UILabel *genreLabel;
@property (nonatomic, weak) IBOutlet UIButton *priceButton;
@end
```

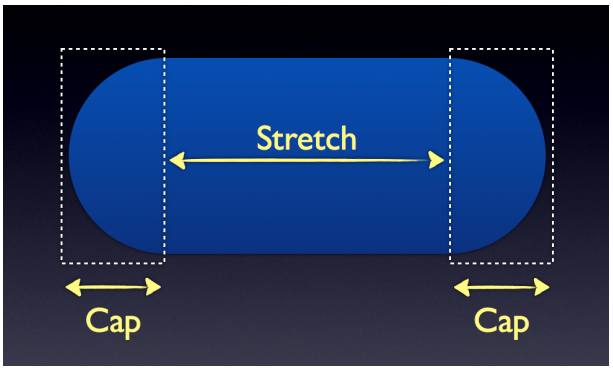
› Connect these properties to the views in the nib. Ctrl-drag from File's Owner to each of the views and pick the corresponding outlet. (The Type: and Genre: labels do not get an outlet.)

➤ Run the app to see if everything still works.



The new controls in the Detail pop-up

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



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

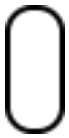
When an image view is wider than the image, it will automatically stretch the image to fit. In the case of a button, however, you don't want to stretch the ends (or "caps") of the button, only the middle part. That's what a stretchable image lets you do.

➤ Change **DetailViewController.m**'s `viewDidLoad` method to:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
```

```
  UIImage *image = [[UIImage imageNamed:@"PriceButton"]
      resizableImageWithCapInsets:UIEdgeInsetsMake(0, 5, 0, 5)];

  [self.priceButton setBackgroundImage:image
                              forState:UIControlStateNormal];
}
```

First this loads an image from the asset catalog with `[UIImage imageNamed:]`. If you look at **PriceButton.png** you will see that it is only 11 pixels wide. That means it has a 5-pixel cap on the left, a 5-pixel cap on the right, and a 1-pixel body that will be stretched out. (The caps on the **@2x.png** version are 10 pixels, of course, because on Retina displays everything is exactly twice as big.)



The PriceButton@2x.png file

`UIImage` has a method `resizableImageWithCapInsets:` that creates a new `UIImage` object that will automatically stretch itself to whatever size it needs to be. It is this stretchable image that you set as the button's background.

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

The price button with the stretchable background image

There main reason you're using a stretchable image here is that the text on the button may vary in size so you don't know in advance how big the button needs to be. If your app has a lot of custom buttons, then it's also 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. Recall

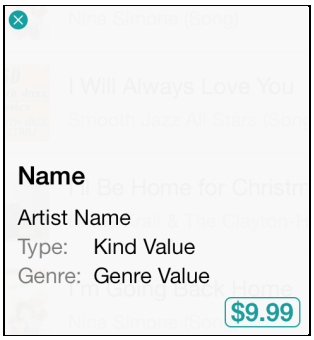
that the color of the button text comes from the global tint color. `UIImage` makes it very easy to make images appear in the same tint color.

➤ Change `viewDidLoad` to:

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  UIImage *image = [[UIImage imageNamed:@"PriceButton"]
     resizableImageWithCapInsets:UIEdgeInsetsMake(0, 5, 0, 5)];

  image = [image imageWithRenderingMode:
                              UIImageRenderingModeAlwaysTemplate];

  [self.priceButton setBackgroundImage:image
                               forState:UIControlStateNormal];

  self.view.tintColor = [UIColor colorWithRed:20/255.0f
                    green:160/255.0f blue:160/255.0f alpha:1.0f];
}
```

When you set the "always template" rendering mode on an image, UIKit will remove the original colors from the image and paint the whole thing in the tint color.

I like the dark green tint color in the rest of the app but for this particular popup it's a bit too dark. You can change the tint color on a per-view basis, and if that view has subviews then the new tint color also applies to these subviews. That's why you set the new `tintColor` on `self.view`, not on `self.priceButton`. Now the lighter tint color applies 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 can use an image to make it look like that but instead I'll show you a little trick.

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

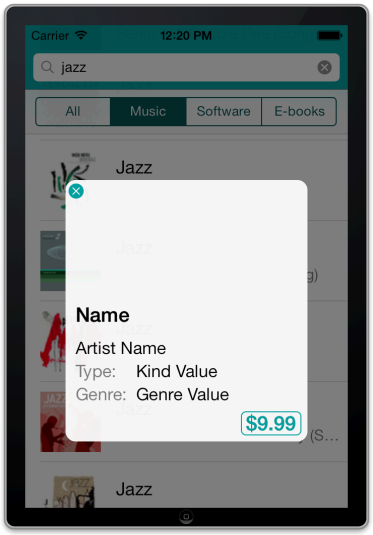
➤ Add the following line to `viewDidLoad`:

```
self.popupView.layer.cornerRadius = 10.0f;
```

You ask the Popup View for its layer and then set the corner radius of that layer to 10 points. And that's all you need to do! Well, you do need add an import. Because `CALayer` is not part of UIKit but the QuartzCore framework, you have to import the headers for this framework:

```
#import <QuartzCore/QuartzCore.h>
```

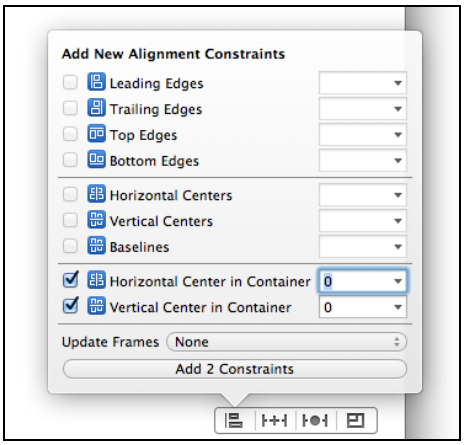
➤ Run the app. There's your rounded corners:



The pop-up now has rounded corners

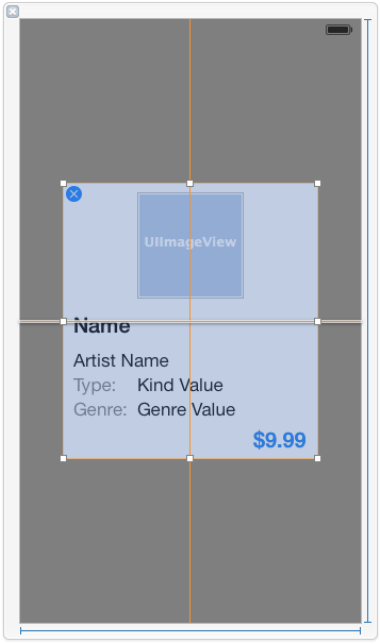
The pop-up still appears off-center on the 3.5-inch Simulator but in the nib it looks fine. That's because the dimensions of the nib are 320×568 points – the size of a 4-inch device – but on 3.5-inch there are only 480 vertical points. So you need to compensate for that somehow. Auto Layout to the rescue!

➤ Open **DetailViewController.xib** and select the **Popup View**. Click the **Align** button at the bottom of the canvas, and put checkmarks in front of **Horizontal Center in Container** and **Vertical Center in Container**.



Adding constraints to align the Popup View

› Press **Add 2 Constraints** to finish. This adds two new constraints to the Popup View, represented by the orange lines that cross the nib:



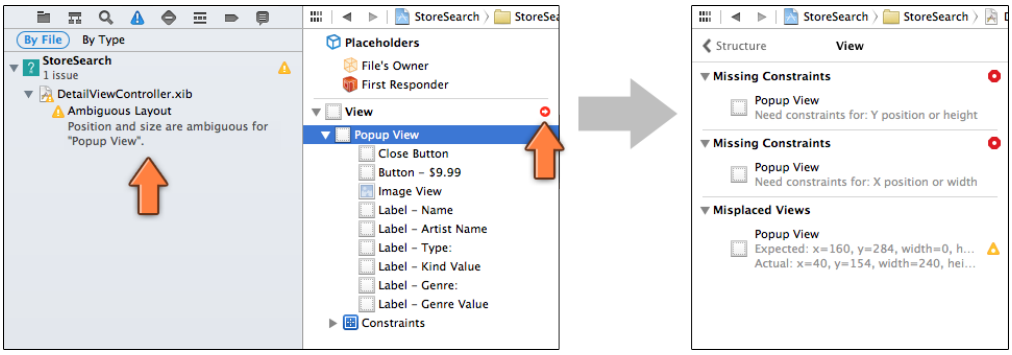
The Popup View with alignment constraints

One small hiccup: these lines are supposed to be blue, not orange. Whenever you see 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 Popup View, based on where you placed that view in Interface Builder. But as soon as you add a single constraint of your own, you no longer get these automatic constraints.

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

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



Xcode shows Auto Layout errors in the Issue navigator

➤ Tap the small red arrow in the outline pane to get a more detailed explanation of the errors. It's obvious that something's missing. You know it's not the position – the two alignment constraints are enough to determine that – so it must be the size.

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

➤ Select the Popup View and click the **Pin** button. Put checkmarks in front of **Width** and **Height**. Click **Add 2 Constraints** to finish.

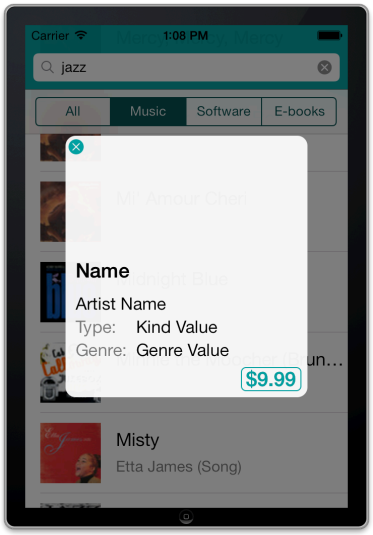
Pinning the width and height of the Popup View

Now the lines turn blue and Auto Layout is happy. However, there is still one step left. After you instantiate the `DetailViewController` it always has a view that is 568 points high, even on a 3.5-inch device. Before you add its view to the window you need to resize it to the proper dimensions.

➤ In **SearchViewController.m**, in `didSelectRowAtIndexPath` add the following line just before `[self.view addSubview:…]`:

```
controller.view.frame = self.view.frame;
```

➤ Run the app on the 3.5-inch and 4-inch Simulators and verify that the popup now always shows up in the exact center of the screen:



The Detail popup is now centered in the screen

The close button is pretty small, about 15 by 20 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 popup by tapping anywhere outside it. The ideal tool for this job is a **gesture recognizer**.

➤ Change the `@interface` line inside **DetailViewController.m** to:

```
@interface DetailViewController () <UIGestureRecognizerDelegate>
```

➤ Add the following lines to `viewDidLoad`:

```
UITapGestureRecognizer *gestureRecognizer =
     [[UITapGestureRecognizer alloc] initWithTarget:self
                                     action:@selector(close:)];

gestureRecognizer.cancelsTouchesInView = NO;
gestureRecognizer.delegate = self;
```

```
[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. Of course, you only want to close the popup when the user taps outside the popup, i.e. on the background. Any other taps should be ignored. That's what the delegate method is for.

▸ Add the following method below `viewDidLoad:`

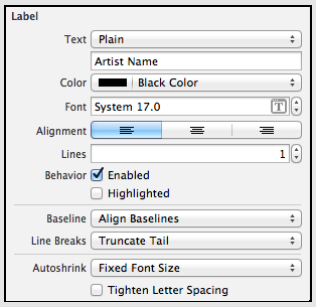
```
- (BOOL)gestureRecognizer:
           (UIGestureRecognizer *)gestureRecognizer
        shouldReceiveTouch:(UITouch *)touch
{
  return (touch.view == self.view);
}
```

This only returns YES when the touch was on the background view but NO if it was inside the Popup View.

▸ Try it out. You can now dismiss the popup by tapping anywhere outside the white popup 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!

Attributes and properties

Most of the attributes in Interface Builder's inspectors correspond directly to properties on the selected object. For example, a `UILabel` has the following attributes:

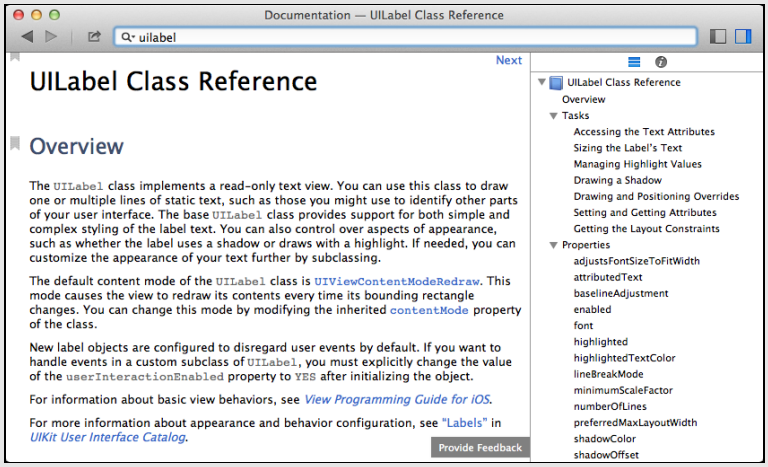


These are directly related to the following properties:

Text	`label.text`
Lines	`label.numberOfLines`
Behavior: Enabled	`label.enabled`
Baseline	`label.baselineAdjustment`
Line Breaks	`label.lineBreakMode`
Alignment	`label.textAlignment`

And so on... As you can see, the names may not always be exactly the same ("Lines" and `numberOfLines`) but you can easily figure out which property goes with which attribute.

You can find the properties in the documentation for `UILabel`. From the **Help** menu, select **Documentation and API Reference** to open the documentation window. Type "uilabel" into the search field to bring up the class reference for `UILabel`:



The documentation for `UILabel` may not list properties for all of the attributes from the inspectors. For example, in the Attributes inspector there is a section named "View". The attributes in this section come from `UIView`, which is the base class of `UILabel`. So if you can't find a property in the `UILabel` class, you may need to check the documentation of its superclasses.

Putting the data into the Detail pop-up

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

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

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

➤ Add this property to **DetailViewController.h**, as well as a forward declaration for the `SearchResult` class:

```
@class SearchResult;

. . .
```

```
@property (nonatomic, strong) SearchResult *searchResult;
```

Unlike outlet properties – which you want to keep hidden from the rest of the world – this property should be in the public header file because you need an other object to set it (namely the `SearchViewController`).

The place where you launch this pop-up view is in `SearchViewController`'s `didSelectRowAtIndexPath`. Conveniently enough, in that method you also have the index-path of the selected row, which lets you find the `SearchResult` object. Put two and two together and you'll end up with:

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  [tableView deselectRowAtIndexPath:indexPath animated:YES];

  DetailViewController *controller = [[DetailViewController
     alloc] initWithNibName:@"DetailViewController" bundle:nil];

  SearchResult *searchResult = _searchResults[indexPath.row];
  controller.searchResult = searchResult;

  controller.view.frame = self.view.bounds;
  [self.view addSubview:controller.view];
  [self addChildViewController:controller];
  [controller didMoveToParentViewController:self];
}
```

It's mostly the same as before, except you now look up the `SearchResult` object and put it in `DetailViewController`'s property.

> **Note:** Again, the order of operations is important here. You are going to read from the `searchResult` property in `DetailViewController`'s `viewDidLoad` method. That only works if you set the property before the view gets loaded. But when exactly does the view get loaded?
>
> That happens the first time you (or anyone else) access the new view controller's `view` property. And that's exactly what the next line does when it sets the frame with `controller.view.frame`. If you were set the `searchResult` property after that line, then `viewDidLoad` already happened and the Detail screen stays empty.

▸ In **DetailViewController.m**, first add the import for the `SearchResult` class:

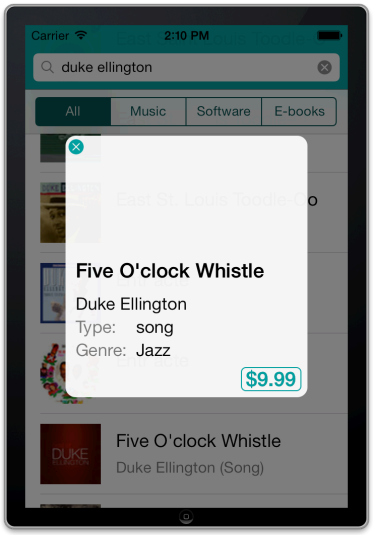
```
#import "SearchResult.h"
```

▸ Change `viewDidLoad` to:

```
- (void)viewDidLoad
{
  . . .

  if (self.searchResult != nil) {
    [self updateUI];
  }
}

- (void)updateUI
{
  self.nameLabel.text = self.searchResult.name;

  NSString *artistName = self.searchResult.artistName;
  if (artistName == nil) {
    artistName = @"Unknown";
  }

  self.artistNameLabel.text = artistName;
  self.kindLabel.text = self.searchResult.kind;
  self.genreLabel.text = self.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`.

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



The pop-up with filled-in data

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

You could simply copy-paste the `kindForDisplay:` method but then you have identical code in two different places in the app. What if you decide to support another type of product, then you'd have to remember to update this method in two places as well. That sort of thing becomes a maintenance nightmare and is best avoided. Instead, you should look for a better place to put that method.

Exercise: Where would you put it? □

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

▸ Cut the `kindForDisplay:` method out of the `SearchResultCell` source code and put it in **SearchResult.m**.

Because `SearchResult` already has a `kind` property, you no longer have to pass that value as a parameter to this method but you can simply use `self.kind` instead.

▸ Change the method to:

```
- (NSString *)kindForDisplay
{
  if ([self.kind isEqualToString:@"album"]) {
    return @"Album";
  } else if ([self.kind isEqualToString:@"audiobook"]) {
    return @"Audio Book";
  } else if ([self.kind isEqualToString:@"book"]) {
    return @"Book";
  } else if ([self.kind isEqualToString:@"ebook"]) {
    return @"E-Book";
  } else if ([self.kind isEqualToString:@"feature-movie"]) {
    return @"Movie";
  } else if ([self.kind isEqualToString:@"music-video"]) {
    return @"Music Video";
  } else if ([self.kind isEqualToString:@"podcast"]) {
    return @"Podcast";
  } else if ([self.kind isEqualToString:@"software"]) {
    return @"App";
  } else if ([self.kind isEqualToString:@"song"]) {
    return @"Song";
  } else if ([self.kind isEqualToString:@"tv-episode"]) {
    return @"TV Episode";
```

```
  } else {
    return self.kind;
  }
}
```

➤ Also add a method signature to **SearchResult.h**, so other classes will know of the existence of this method:

```
- (NSString *)kindForDisplay;
```

Of course, **SearchResultCell.m** now tries to call a method that no longer exists, so you have to fix it to use the `SearchResult` object instead.

➤ Change its `configureForSearchResult:` method to now do:

```
NSString *kind = [searchResult kindForDisplay];
```

➤ Let's also do that in **DetailViewController.m**. Change its `updateUI` method to do:

```
self.kindLabel.text = [self.searchResult kindForDisplay];
```

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

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

It's OK to start out with your code being a bit of a mess (that's what it often is for me!) but whenever you see an opportunity to clean things up and simplify it, you should take it. As your source code evolves, it will become clearer to you what the best internal structure is for that particular program. But you have to be willing to revise the code when you realize it can be improved in some way!

There are three more things to do on this screen:

1. Show the price, in the proper currency.
2. Make the price button work so it opens the product page in the iTunes store.
3. Download and show the artwork image. This image is slightly larger than the one from the table view cell.

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

➤ Add the following code to `updateUI`:

```
NSNumberFormatter *formatter = [[NSNumberFormatter alloc] init];
```

```
[formatter setNumberStyle:NSNumberFormatterCurrencyStyle];
[formatter setCurrencyCode:self.searchResult.currency];

NSString *priceText;
if ([self.searchResult.price floatValue] == 0.0f) {
  priceText = @"Free";
} else {
  priceText = [formatter
                      stringFromNumber:self.searchResult.price];
}

[self.priceButton setTitle:priceText forState:UIControlStateNormal];
```

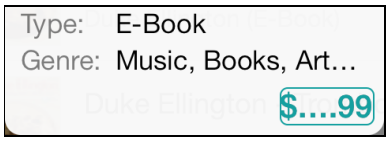
You've used `NSDateFormatter` in previous tutorials to turn an `NSDate` object into human-readable text. Here you use `NSNumberFormatter` to do the same thing for numbers. It's possible, of course, to turn a number into a string using `stringWithFormat:` and the right format specifier such as `@"%f"` or `@"%d"`. 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 `NSNumberFormatter` to do all the hard work.

You simply tell the `NSNumberFormatter` 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". The `NSNumberFormatter` will insert the proper symbol, such as $ or € or ¥, and formats the monetary amount according to the user's regional settings.

➤ Run the app and see if you can find some good deals. ☺

Occasionally you might see this:



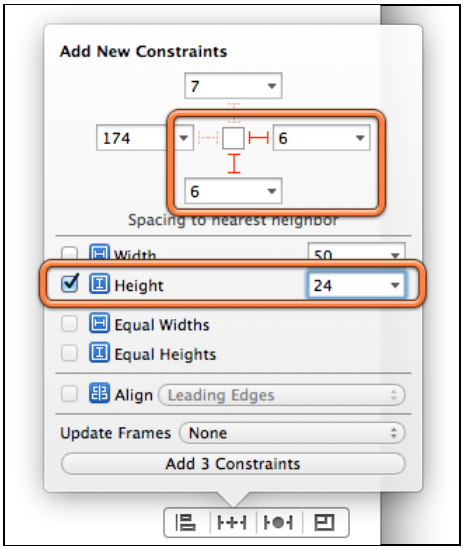
The price doesn't fit into the button

When you designed the nib you made this button 60 points wide. You didn't put any constraints on it, so Xcode gave it an automatic constraint that always forces the button to be 60 points wide. But buttons, like labels, are perfectly able to determine what their ideal size is based on the amount of text they contain. That's called the **intrinsic content size**.

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

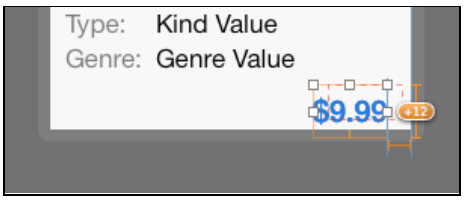
That alone is not enough. You also need to add at least one constraint to the button or Xcode will still apply the automatic constraints.

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



Pinning the price button

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



Orange bars indicate the button is misplaced

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

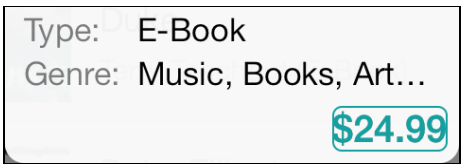
› Select the button and from the menu bar choose **Editor → Resolve Auto Layout Issues → Update Frames**. Now the lines should all turn blue.

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

- Fixed height of 24 points. That is necessary because the background image is 24 points tall.

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

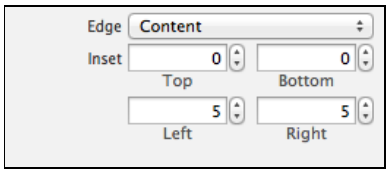
➤ Run the app again and pick an expensive product (something with a price over $9.99; e-books are a good category for this).



The button is a little cramped

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

➤ Go to the **Attributes inspector** and find where it says **Edge: Content**. Change **Left** and **Right** to 5.

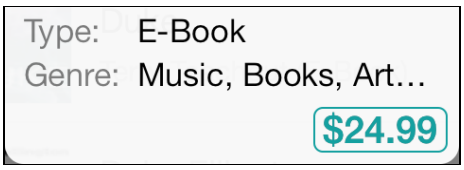


Changing the content edge insets of the button

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

➤ Once again, choose **Update Frames** from the **Resolve Auto Layout Issues** menu.

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



It looks so good you almost want to tap it!

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

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

```
- (IBAction)openInStore:(id)sender
{
  [[UIApplication sharedApplication] openURL:
              [NSURL URLWithString:self.searchResult.storeURL]];
}
```

➤ And connect it to the button's Touch Up Inside event (in the nib).

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

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

➤ First add the import:

```
#import <AFNetworking/UIImageView+AFNetworking.h>
```

➤ Then add the following line to `updateUI`:

```
[self.artworkImageView setImageWithURL:
         [NSURL URLWithString:self.searchResult.artworkURL100]];
```

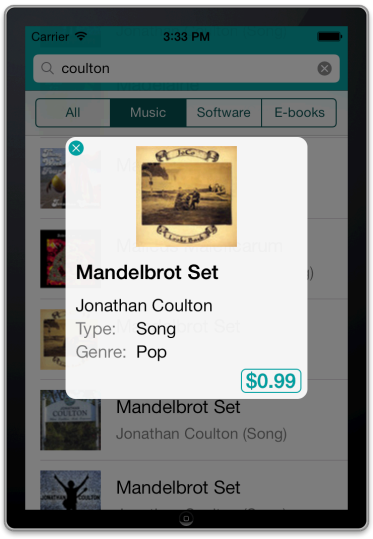
This is the same thing you did in `SearchResultCell`, except that you use the other artwork URL (100×100 pixels) and no placeholder image.

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

➤ Change `dealloc` to:

```
- (void)dealloc
{
  NSLog(@"dealloc %@", self);

  [self.artworkImageView cancelImageRequestOperation];
}
```

➤ Try it out!



The pop-up now shows the artwork image

➤ This is a good time to commit the changes.

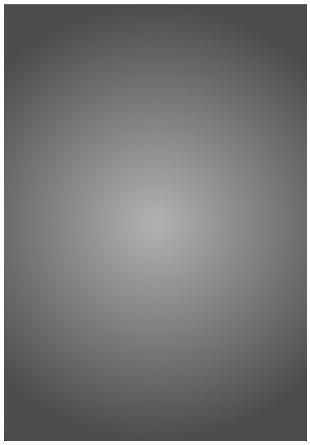
Gradients in the background

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

It works well, but on the other hand, a plain black overlay is a bit dull. Let's turn it into a circular gradient instead. You could use Photoshop to draw such a gradient and place an image view behind the popup, but why use an image when you can also draw using Core Graphics? To pull this off, you will create your own `UIView` subclass.

➤ Add a new file to the project, **Objective-C class** template. Name it **GradientView** and make it a subclass of **UIView**.

This will be a very simple view. It simply draws a black circular gradient that goes from a mostly opaque in the corners to mostly transparent in the center. It's a similar dimmed background that you used to see behind `UIAlertViews` on iOS 6 and earlier. Placed on a white background, it looks like this:



What the GradientView looks like by itself

▸ Replace the contents of **GradientView.m** by:

```
#import "GradientView.h"

@implementation GradientView

- (id)initWithFrame:(CGRect)frame
{
  if ((self = [super initWithFrame:frame])) {
    self.backgroundColor = [UIColor clearColor];
  }
  return self;
}

- (void)drawRect:(CGRect)rect
{
  // 1
  const CGFloat components[8] = { 0.0f, 0.0f, 0.0f, 0.3f,
                                  0.0f, 0.0f, 0.0f, 0.7f };
  const CGFloat locations[2] = { 0.0f, 1.0f };

  // 2
  CGColorSpaceRef space = CGColorSpaceCreateDeviceRGB();
  CGGradientRef gradient = CGGradientCreateWithColorComponents(
                               space, components, locations, 2);
  CGColorSpaceRelease(space);

  // 3
  CGFloat x = CGRectGetMidX(self.bounds);
  CGFloat y = CGRectGetMidY(self.bounds);
  CGPoint point = CGPointMake(x, y);
  CGFloat radius = MAX(x, y);
```

```
  // 4
  CGContextRef context = UIGraphicsGetCurrentContext();
  CGContextDrawRadialGradient(context, gradient, point, 0,
              point, radius, kCGGradientDrawsAfterEndLocation);

  // 5
  CGGradientRelease(gradient);
}

@end
```

In the init method you simply set the background color to fully transparent. Then in `drawRect:` you draw the gradient on top of that transparent background, so that it blends with whatever is below.

The drawing code uses the Core Graphics framework (also known as Quartz 2D). It may look a little scary but this is plain old C code.

1. First you create two C-style arrays that contain the "color stops" for the gradient. The first color (0.0, 0.0, 0.0, 0.3) is a black color that is mostly transparent. It sits at position 0.0 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, 0.0, 0.7) is also black but much less transparent and sits at location 1.0, 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 from 0.0 to 1.0.)

2. With those color stops you can create the gradient. This gives you a new `CGGradientRef` object. This is not an object in the sense of Objective-C, so you cannot send it messages by doing `[gradient methodName]`. But it's still some kind of data structure that lives in memory, and the `gradient` variable points to it.

3. Now that you have the gradient object, you have to figure out how big you need to draw it. The `CGRectGetMidX()` and `CGRectGetMidY()`functions return the center point of a rectangle. That rectangle is given by `self.bounds`, which tells you how big the view is.

   If I can avoid it, I prefer not to hard-code any dimensions such as "320 by 480 points". By asking `self.bounds`, you can use this view anywhere you want to, no matter how big a space it should fill (you could reuse it without problems on an iPad, for example, whose screen is a lot bigger!).

   The `point` variable 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 macro that you can use to determine which of two values is the biggest.

4. With all those preliminaries done, you can finally draw the thing. Core Graphics drawing always takes places in a so-called 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. The `CGContextDrawRadialGradient()` function finally draws the gradient according to your specifications.

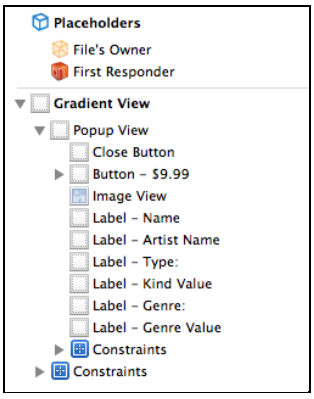
5. When you're done, you need to release the gradient object (notice that you also released the `CGColorSpaceRef` object earlier).

> **Note:** I have mentioned manual memory management a few times now and how that is no longer necessary since the arrival of ARC. The caveat there is that Automatic Reference Counting only works on Objective-C code, not on plain old C code. And that's exactly what Core Graphics is, plain old C code. Any time you create a new object with Core Graphics, such as the `CGGradientRef` object you made with `CGGradientCreateWithColorComponents()`, you also have to release it once you're done with it.
>
> So even with ARC you cannot always escape some manual memory management duties. If you forget to call `CGGradientRelease()` here, you would leak an object every time the user opens the pop-up. A gradient object may only take up a few bytes of storage space but all those small leaks over time add up to a big one.

It generally speaking isn't optimal to create new objects inside your `drawRect:` method, especially if `drawRect:` is called often. In that case it is better to create the objects the first time you need them, and to reuse the same instance over and over (lazy loading!). You don't really have to do that here because this `drawRect:` method will be called just once – when the `DetailViewController` gets loaded – so you can get away with being less than optimal.

Using this new `GradientView` class is pretty easy.

› Open **DetailViewController.xib** and select the nib's main view. That's the view you previously made 50% black. Go to the **Identity inspector** and change its class from UIView to **GradientView**.



The top-level view is now a GradientView

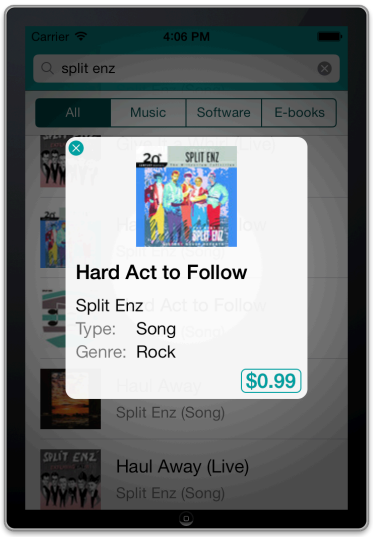
From now on when the nib gets loaded, the top-level view from `DetailViewController` (the one you access with `self.view`) is no longer a regular `UIView` class but your very own `GradientView`.

There's one more thing to do: because the view's 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 that makes it harder to see the pop-up inside the nib. So let's do that in code instead.

➤ Add the following line the **DetailViewController.m**'s `viewDidLoad`:

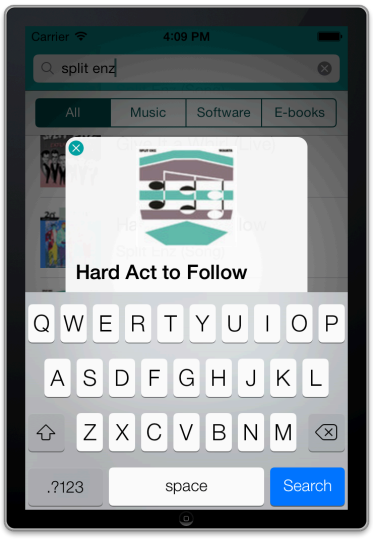
```
self.view.backgroundColor = [UIColor clearColor];
```

➤ Run the app and see what happens. Nice, that looks a lot smarter.



The background behind the pop-up now has a gradient

You're not out of the woods yet. It's always a good idea to test your app for weird things users might do that you didn't expect. What if, after performing a search, the user taps the search bar again – thereby putting the keyboard back on the screen – and then selects a row? This is what:



The keyboard obscures the pop-up

➤ Fortunately this is easily fixed. Add the following line to the top of `didSelectRowAtIndexPath` in **SearchViewController.m**:

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  [self.searchBar resignFirstResponder];

  . . .
}
```

Now any visible keyboard will slide out of view first.

➤ You're going to make some big changes in the next section, so commit your progress.

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

Animation!

The pop-up itself looks good already, but the way it enters the screen – Poof! It's suddenly there – isn't very exciting and possibly even a bit unsettling. iOS is the king of animation, so let's make good on that.

You've used Core Animation before. In the Bull's Eye tutorial you used `CATransition` to achieve a cross-fade effect, in MyLocations you used a `CABasicAnimation` to rotate and move a view, and this time you'll use a so-called **keyframe animation** to make the pop-up bounce into view.

First you will refactor the code a little. Currently the code to present the pop-up is part of `SearchViewController` and the code to dismiss it is in `DetailViewController`. I want to put both pieces of logic into the same class.

➤ Add the following method signatures to **DetailViewController.h**:

```
- (void)presentInParentViewController:
                        (UIViewController *)parentViewController;

- (void)dismissFromParentViewController;
```

You already have the logic for `dismissFromParentViewController` but it's presently in the `close:` action method in **DetailViewController.m**. Rewrite this code to:

```
- (IBAction)close:(id)sender
{
  [self dismissFromParentViewController];
}

- (void)dismissFromParentViewController
{
  [self willMoveToParentViewController:nil];
  [self.view removeFromSuperview];
  [self removeFromParentViewController];
}
```

The result is the same but I prefer to have the logic inside its own method. (Later in this tutorial you'll be using `dismissFromParentViewController` from another place as well.)

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

```
- (void)presentInParentViewController:
                         (UIViewController *)parentViewController
{
  self.view.frame = parentViewController.view.bounds;
  [parentViewController.view addSubview:self.view];
  [parentViewController addChildViewController:self];
  [self didMoveToParentViewController:parentViewController];
}
```

This does what you used to do before in `SearchViewController` but the other way around. It is now accomplished from the point of view of the `DetailViewController` (the child) rather than the `SearchViewController` (the parent).

➤ Change **SearchViewController.m**'s `didSelectRowAtIndexPath` method to:

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  [self.searchBar resignFirstResponder];
  [tableView deselectRowAtIndexPath:indexPath animated:YES];

  DetailViewController *controller = [[DetailViewController
     alloc] initWithNibName:@"DetailViewController" bundle:nil];

  SearchResult *searchResult = _searchResults[indexPath.row];
  controller.searchResult = searchResult;

  [controller presentInParentViewController:self];
}
```

That should do it. Run the app to make sure everything still works as it should.

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

➤ Change `presentInParentViewController:` to:

```
- (void)presentInParentViewController:
                        (UIViewController *)parentViewController
{
  self.view.frame = parentViewController.view.bounds;
  [parentViewController.view addSubview:self.view];
  [parentViewController addChildViewController:self];

  CAKeyframeAnimation *bounceAnimation = [CAKeyframeAnimation
                       animationWithKeyPath:@"transform.scale"];

  bounceAnimation.duration = 0.4;
  bounceAnimation.delegate = self;

  bounceAnimation.values = @[ @0.7, @1.2, @0.9, @1.0 ];
  bounceAnimation.keyTimes = @[ @0.0, @0.334, @0.666, @1.0 ];

  bounceAnimation.timingFunctions = @[
    [CAMediaTimingFunction
          functionWithName:kCAMediaTimingFunctionEaseInEaseOut],
    [CAMediaTimingFunction
          functionWithName:kCAMediaTimingFunctionEaseInEaseOut],
    [CAMediaTimingFunction
         functionWithName:kCAMediaTimingFunctionEaseInEaseOut]];

  [self.view.layer addAnimation:bounceAnimation
                         forKey:@"bounceAnimation"];
```

```
}
```

After adding the view and view controller to the parent view controller, you create a `CAKeyframeAnimation` object that works on the view's `transform.scale` attributes. That means you'll be animating the size of the view.

The animation consists of several **keyframes**. It will smoothly proceed from one keyframe to the other over a certain amount of time. The values for the keyframes are added to the animation's `values` property. Because you're animating the view's scale, these particular values represent how much bigger or smaller the view will be over time.

The animation starts with the view scaled down to 70% (scale 0.7). The next keyframe inflates it to 120% its normal size. After that, it will scale the view down a bit again but not as much as before (only 90% of its original size), and the last 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.

For each keyframe you can also specify a time. You can see this as the duration between two successive keyframes. In this case, each transition from one keyframe to the next takes 1/3rd of the total animation time. Note that these times are not in seconds but in fractions of the animation's duration. The total duration is 0.4 seconds.

You can also specify a **timing function** that is used to go from one keyframe to the next. I chose to use the "Ease In, Ease Out" function, which starts slowly, then ramps up to full speed, and then slows down again. You can also choose just "Ease In" (think accelerating) or "Ease Out" (think decelerating) or a linear interpolation, but the latter doesn't tend to look very realistic. Moving objects in real-life always need some time to get started or slow down and that is what the timing function does.

Finally, you add the animation to the view's layer. Core Animation doesn't work on the `UIView` objects themselves but on their `CALayer`s.

You also set the animation's delegate. You need to know when the animation stops because at that point you must call the `didMoveToParentViewController:` method. That's part of the three steps of embedding one view controller into another, remember? By making `DetailViewController` the delegate of the `CAKeyframeAnimation`, you will be told when the animation stopped.

➤ Add the following method:

```
- (void)animationDidStop:(CAAnimation *)anim finished:(BOOL)flag
{
  [self didMoveToParentViewController:
```

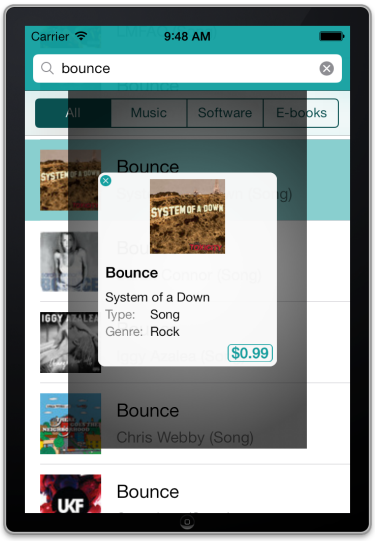
```
                                    self.parentViewController];
}
```

The `addChildViewController:` method that puts one view controller "inside" another will automatically call the `willMoveToParentViewController:` method on the child view controller to let it know it's about to be added to a parent controller. However, it cannot also automatically call `didMoveToParentViewController:`, as that method needs to happen after any animations that you do. So you're always responsible for calling this method on the child view controller once the animation completes. Don't forget this; it's part of the rules for embedding view controllers.

Note that for this Core Animation delegate there is no need to put a protocol declaration into the `@interface` line. `CAKeyframeAnimation` doesn't use a formal protocol for its delegate. All you need to do is implement the `animationDidStop:finished:` method and you're done. (If you want, you could put an `NSLog()` in this method to verify that it's being called. It's always a good idea to make sure that your code does what you expect it to!)

Oh, in case you were wondering where the `self.parentViewController` property came from: that is defined in `UIViewController`. When a view controller has been added to a parent controller using `addChildViewController:`, its parent property points to that parent controller. That way you'll always know who to call when you're in trouble.

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



The popup animates, but so does the gradient background

W00t, that bounced a bit too much for my taste. I like the way the pop-up bounces into the screen, but did you notice that the `GradientView` also bounced along with it? Since the `GradientView` is part of the `DetailViewController`, it becomes part of the animation as well!

Exercise. Think of a way to fix this. □

Answer: There are two solutions I can come up with:

1. Put the animation on the `popupView`'s layer instead of `self.view` so that it doesn't bounce the whole thing but only the square that contains the labels.
2. Don't use the `GradientView` as the main view for `DetailViewController` but add it as a separate subview to the parent view controller. Let's pick this option.

➤ Open **DetailViewController.xib** and in the **Identity inspector** set the **Class** of the main view back to **UIView**" (You can simply clear out the text box and the class will revert to "UIView" automatically.)

Putting `GradientView` here sounded like a good idea but it didn't work so well once you started animating.

➤ You're going to be working with the `GradientView` class inside **DetailViewController.m**, so first add an import:

```
#import "GradientView.h"
```

➤ Add a new instance variable that will hold the `GradientView` object:

```
@implementation DetailViewController
{
  GradientView *_gradientView;
}
```

➤ Add the following lines to the top of `presentInParentViewController:`

```
- (void)presentInParentViewController:
                        (UIViewController *)parentViewController
{
  _gradientView = [[GradientView alloc]
                initWithFrame:parentViewController.view.bounds];
  [parentViewController.view addSubview:_gradientView];

  . . .
```

You create a new `GradientView` object by hand that is just as big as the view from the parent view controller. Then you add it as a subview to that parent view controller. Notice that you're doing this before you add `DetailViewController`'s view to the parent view controller, which causes the `GradientView` to sit below the pop-up, which is exactly where you want it.

➤ Run the app. The only thing that is bouncing now should be the pop-up. Much nicer!

Of course, fixing one problem leads to another. When you now tap the close button, the pop-up disappears but the gradient view doesn't. Since it lies on top of the

table view and all other controls, the app doesn't react to any input anymore. Because you added this `GradientView` object when you invoked the pop-up, you also have to remove it again when the pop-up gets dismissed.

➤ Change `dismissFromParentViewController` to:

```
- (void)dismissFromParentViewController
{
  . . .

  [_gradientView removeFromSuperview];
}
```

That's better. If you want to make sure that the `GradientView` object gets deallocated properly, you can give it a `dealloc` method with an `NSLog()` statement. I'm paranoid about such things – by looking at the code you may reason that the object should be deleted, but putting in an `NSLog()` will tell you for sure.

➤ Add to **GradientView.m**, and verify that the gradient view is properly deallocated:

```
- (void)dealloc
{
  NSLog(@"dealloc %@", self);
}
```

The pop-up looks a lot spiffier with the animations but there are two things that could be better: the `GradientView` still appears abruptly in the background, and there is no animation on dismissal of the pop-up.

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.

➤ Add the following code to the bottom of `presentInParentViewController:`

```
CABasicAnimation *fadeAnimation = [CABasicAnimation
                                   animationWithKeyPath:@"opacity"];
fadeAnimation.fromValue = @0.0f;
fadeAnimation.toValue = @1.0f;
fadeAnimation.duration = 0.2;
[_gradientView.layer addAnimation:fadeAnimation
                           forKey:@"fadeAnimation"];
```

This is a simple `CABasicAnimation` that animates the `_gradientView`'s opacity value from 0.0 (fully see-through) to 1.0 (fully visible) in 0.2 seconds, resulting in a simple fade-in. That's a bit more subtle than making the view appear so abruptly.

To make the pop-up disappear, you'll use a basic `UIView` animation block.

➤ Replace `dismissFromParentViewController` with:

```
- (void)dismissFromParentViewController
{
  [self willMoveToParentViewController:nil];

  [UIView animateWithDuration:0.3 animations:^{
    CGRect rect = self.view.bounds;
    rect.origin.y += rect.size.height;
    self.view.frame = rect;
    _gradientView.alpha = 0.0f;

  } completion:^(BOOL finished) {
    [self.view removeFromSuperview];
    [self removeFromParentViewController];

    [_gradientView removeFromSuperview];
  }];
}
```

The order of operations remains the same, except now there's a short animation before the view and the view controller truly get removed from the parent controller.

You're making the pop-up slide down the screen. To do that, you change its **frame**. The frame is the rectangle that describes the view's position and size in terms of its superview. When the pop-up is fully visible, its frame has coordinates (0, 0) – i.e. the top-left corner of the screen – and size 320×480 (or 568 on 4-inch devices). In this animation block you will move the view to coordinates **(0, height)**, which puts it just below the screen where it is no longer visible. This creates an effect of the view dropping out of sight.

You also animate the gradient view by setting its alpha value to 0, which will fade it out. That's essentially the reverse of what you did earlier except that the property is now called "alpha" instead of "opacity" (that's one of the small differences between doing `UIView`-based animation and Core Animation; you gotta love it).

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

Note: The rules for view controller containment, which you can find in the documentation for `UIViewController`, say that you shouldn't call `didMoveToParentViewController:` when you're removing the child view controller, as `removeFromParentViewController` will already do that, but you do need to call `willMoveToParentViewController:` on the child view controller to let it know it's about to be removed from its parent. You don't actually remove

the child controller until the animation completes. This order of operations is exactly the other way around from adding a view controller.

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

You can find the project files for this section under **07 - Animation** in the tutorial's Source Code folder.

Frame vs. bounds

In the code above, you do the following:

```
CGRect rect = self.view.bounds;
rect.origin.y += rect.size.height;
self.view.frame = rect;
```

First you get the view's bounds, which describe a rectangle, then you change the Y coordinate of that rectangle, and finally you assign this new rectangle to the view's frame. So what is the difference between the bounds and the frame?

The frame describes the position and size of a view in its parent view. If you want to put a 50×50 button at position X: 100, Y: 30, then its frame is (100, 30, 50, 50). To move a view from one position to another, you need to change its frame (or its `center` property, which in turn will modify the frame).

Where the frame basically describes the outside of the view, the bounds describe the inside. The X and Y coordinates of the bounds are (0, 0) and the width and height will be the same as the frame. So for the button above, its bounds are (0, 0, 50, 50). It's a matter of perspective.

Sometimes it makes sense to use the bounds; sometimes you need to use the frame. The frame is actually derived from a combination of properties: the center position of the view, the bounds, and any transform that is set on the view. (Transforms are used for rotating or scaling the view, which you saw in the keyframe animation earlier.)

Fun with landscape

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

The app looks completely different in landscape orientation

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

The to-do list for this section is:

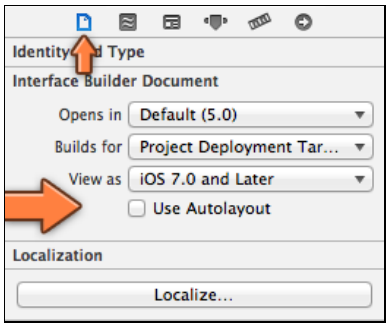
- Create a new view controller and show that when the device is rotated. Hide this view controller when the device returns to the portrait orientation.
- Put some fake buttons in a `UIScrollView`, in order to learn how to use scroll views.
- Add the paging control (the dots at the bottom) so you can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

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

› Add a new file to the project. Name it **LandscapeViewController** and make it a subclass of **UIViewController**. As before this will have a XIB for the user interface.

› In Interface Builder, select the nib's main view and change its **Orientation** to **Landscape** (in the **Attributes inspector**). Also change the **Background** to **Black Color**.

› Disable Auto Layout for this nib. You do this by unchecking the **Use Autolayout** box in the **File inspector**:

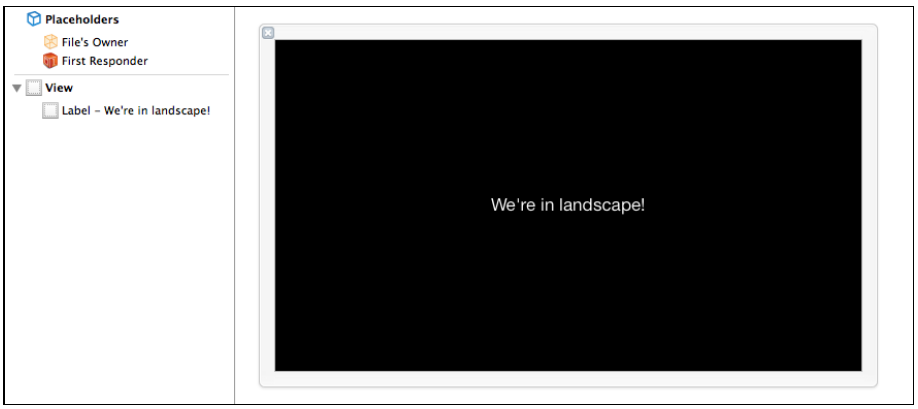


Disabling Auto Layout for this nib

Auto Layout is enabled or disabled on a per-nib basis. For this view controller it's easier to not use Auto Layout and position all the subviews programmatically.

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

Your design should look something like this:



Initial design for the landscape view controller

You should know by now that view controllers have a bunch of methods that are invoked by UIKit at given times, such as `viewDidLoad`, `viewWillAppear:`, and so on. One of these methods is `willRotateToInterfaceOrientation:duration:`. When a view controller is about to be flipped over, it will let you know with this method. You can override it to show (and hide) the new `LandscapeViewController`.

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

```
- (void)willRotateToInterfaceOrientation:
                (UIInterfaceOrientation)toInterfaceOrientation
        duration:(NSTimeInterval)duration
{
  [super willRotateToInterfaceOrientation:toInterfaceOrientation
                                 duration:duration];

  if (UIInterfaceOrientationIsPortrait(
```

```
                                        toInterfaceOrientation)) {
    [self hideLandscapeViewWithDuration:duration];
  } else {
    [self showLandscapeViewWithDuration:duration];
  }
}
```

Just to keep things readable, the actual showing and hiding happens in methods of their own.

▸ First import the new view controller:

```
#import "LandscapeViewController.h"
```

▸ And add an instance variable:

```
LandscapeViewController *_landscapeViewController;
```

I'll tell you why this instance variable is necessary in a moment.

▸ Add the following method:

```
- (void)showLandscapeViewWithDuration:(NSTimeInterval)duration
{
  if (_landscapeViewController == nil) {

    _landscapeViewController = [[LandscapeViewController alloc]
         initWithNibName:@"LandscapeViewController" bundle:nil];

    _landscapeViewController.view.frame = self.view.bounds;

    [self.view addSubview:_landscapeViewController.view];
    [self addChildViewController:_landscapeViewController];
    [_landscapeViewController
                          didMoveToParentViewController:self];
  }
}
```

This is very similar to what you did with the `DetailViewController`. First you create the new view controller object, and then you set the frame size of its view and add it as a subview. Finally, you perform the all-important step of embedding the view controller into the hierarchy so it will get all necessary callback events. This is not doing any animation yet. As always, first get it to work and only then make it look pretty.

▸ Of course, you shouldn't forget the method that will hide the landscape view controller again:

```
- (void)hideLandscapeViewWithDuration:(NSTimeInterval)duration
{
  if (_landscapeViewController != nil) {
    [_landscapeViewController
                        willMoveToParentViewController:nil];
    [_landscapeViewController.view removeFromSuperview];
    [_landscapeViewController removeFromParentViewController];
    _landscapeViewController = nil;
  }
}
```

Nothing you haven't seen before, except that you explicitly set the instance variable to `nil` in order to deallocate the `LandscapeViewController` object now that you're done with it.

▸ In **LandscapeViewController.m**, add a `dealloc` method, just so you can see that the view controller truly gets deallocated:

```
- (void)dealloc
{
  NSLog(@"dealloc %@", self);
}
```

▸ Try it out! Run the app, do a search and flip over your iPhone or the Simulator to landscape (remember: to rotate the Simulator, press ⌘ and the arrow keys).

The 3.5-inch Simulator after flipping to landscape

(Note: If you don't do a search first before rotating to landscape, the keyboard remains visible. You'll fix that shortly. Also, on a 3.5-inch screen the label isn't properly centered but that's OK for now.)

The transition 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 `showLandscapeViewWithDuration:` method to:

```
- (void)showLandscapeViewWithDuration:(NSTimeInterval)duration
{
  if (_landscapeViewController == nil) {
    _landscapeViewController = [[LandscapeViewController alloc]
         initWithNibName:@"LandscapeViewController" bundle:nil];

    _landscapeViewController.view.frame = self.view.bounds;
    _landscapeViewController.view.alpha = 0.0f;

    [self.view addSubview:_landscapeViewController.view];
    [self addChildViewController:_landscapeViewController];

    [UIView animateWithDuration:duration animations:^{
      _landscapeViewController.view.alpha = 1.0f;
    } completion:^(BOOL finished) {
      [_landscapeViewController
                            didMoveToParentViewController:self];
    }];
  }
}
```

You're still doing the same things as before, except now the `LandscapeViewController` starts out completely see-through (`alpha` = 0.0) and slowly fades in while the rotation takes place until it's completely visible (`alpha` = 1.0). Notice that you delay the call to `didMoveToParentViewController:` until the animation is completed.

➤ Likewise for `hideLandscapeViewWithDuration:`

```
- (void)hideLandscapeViewWithDuration:(NSTimeInterval)duration
{
  if (_landscapeViewController != nil) {
    [_landscapeViewController
                        willMoveToParentViewController:nil];

    [UIView animateWithDuration:duration animations:^{
      _landscapeViewController.view.alpha = 0.0f;
    } completion:^(BOOL finished) {
      [_landscapeViewController.view removeFromSuperview];
      [_landscapeViewController removeFromParentViewController];
      _landscapeViewController = nil;
    }];
  }
}
```

This time you fade out the view (back to `alpha` = 0.0). You don't remove the view and the controller until the animation is completely done.

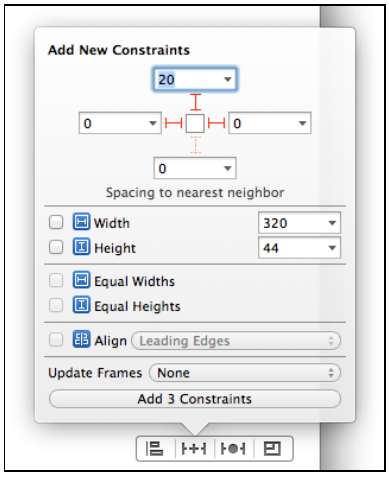
➤ Try it out. The transition between the portrait and landscape views should be a lot smoother now.

If you look closely you'll notice that the main screen is also transformed to landscape dimensions while the landscape view fades in at the same time. This looks a little weird because the search bar and navigation bar don't stretch along with it, creating a gap on the right side of the screen:

There is a gap next to the search bar

It's only a small detail but we can't have such imperfections in our apps! The solution is to pin the search and navigation bars to the edges of the window so that they'll always stretch along with it.

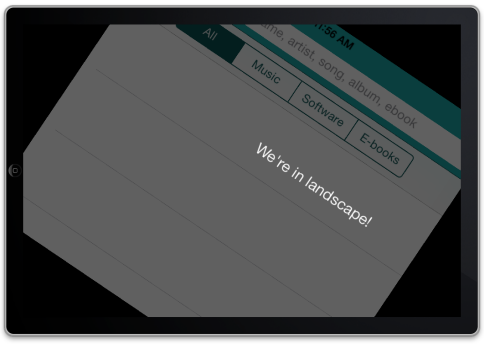
➤ Open **SearchViewController.xib**. Select the Search Bar and pin it to the top, right and left:



Pinning the search bar

➤ Do the same thing for the Navigation Bar. (The top spacing for this should be 0 points because you're pinning the Navigation Bar to the bottom of the Search Bar.)

Now the gap is gone:



The search bar stretches to fill the space

Because the status bar text is black and the landscape screen is also black, the status bar has become invisible. It would be better if it turned white instead.

➤ Add a new instance variable to **SearchViewController.m**:

```
UIStatusBarStyle _statusBarStyle;
```

This variable keeps track of whether the status bar should be black ("default") or white ("light content").

➤ Add the following line to `viewDidLoad` to give this new variable an initial value:

```
_statusBarStyle = UIStatusBarStyleDefault;
```

➤ And add the following method:

```
- (UIStatusBarStyle)preferredStatusBarStyle
{
  return _statusBarStyle;
}
```

This method is called by UIKit to determine what color to make the status bar. You might be thinking that you could put this method into `LandscapeViewController` and make it return `UIStatusBarStyleLightContent` in order to give it a white status bar. Normally that would work, but remember that `LandscapeViewController` is a child view controller that is embedded inside the `SearchViewController`. In this case it is `SearchViewController`'s job to determine the color of the status bar.

➤ Add the following code to the animation block inside `showLandscapeViewWithDuration:`

```
[UIView animateWithDuration:duration animations:^{
  _landscapeViewController.view.alpha = 1.0f;

  _statusBarStyle = UIStatusBarStyleLightContent;
```

```
    [self setNeedsStatusBarAppearanceUpdate];

  } completion:^(BOOL finished) {
      . . .
```

➤ And to the animation block in `hideLandscapeViewWithDuration:`

```
[UIView animateWithDuration:duration animations:^{
  _landscapeViewController.view.alpha = 0.0f;

  _statusBarStyle = UIStatusBarStyleDefault;
  [self setNeedsStatusBarAppearanceUpdate];

} completion:^(BOOL finished) {
    . . .
```

In both cases this changes the value of `_statusBarStyle` and then tells the view controller to update the appearance of the status bar. This causes the `preferredStatusBarStyle` method to be re-evaluated and the status bar will redraw itself in the new color.

Yeah, that looks better:

A white status bar fits the screen much better

So why do you need to use an instance variable to store the `LandscapeViewController` instance? If this variable is not `nil`, that must mean the device is in landscape orientation. When the device rotates back from landscape to portrait, you can remove that `LandscapeViewController` and deallocate it.

But what happens when you rotate the device from "landscape right" to "landscape left" in one go, while skipping the portrait orientation? You don't want to create and show another `LandscapeViewController` on top of the first one.

Try it out. Comment out the `if (landscapeViewController != nil)` statements in both `show–` and `hideLandscapeViewWithDuration:`. Now run the app and flip to landscape mode with ⌘**-Arrow Left**. Then press ⌘**-Arrow Left** two more times.

The first time it will put the phone upside down, an orientation the app doesn't support, so it stays in landscape. The second time it will rotate to the "other" landscape. Now press ⌘**-Arrow Left** one more time to put the Simulator back in portrait. Whoops, that doesn't look too good:

Trust me, we're not in landscape anymore...

You can see in the Xcode Debug pane that the `LandscapeViewController` did `dealloc`, but that was only one of them. When you flipped to the "other" landscape, the app added a new controller on top of the first one and that's the one you're still seeing. Long story short, you prevent against this situation using that instance variable, to make sure you never create more than one `LandscapeViewController` at a time. (Don't forget to put the if-statements back in!)

People may use your apps in ways that you didn't intend or expect, so you need to be one step ahead of them and think through all possible situations the app may have to face. I only discovered that you could rotate from one landscape orientation to the other by accident. Make sure to give your apps a good round of testing and try to "break" them in interesting ways. This will save you from embarrassing bugs!

Hiding the keyboard and pop-up

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

The keyboard is still showing in landscape mode

Exercise. See if you can fix that yourself.

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

▸ Add the following line to `showLandscapeViewWithDuration:`

```
- (void)showLandscapeViewWithDuration:(NSTimeInterval)duration
{
  if (_landscapeViewController == nil) {
    . . .

    [self.searchBar resignFirstResponder];
  }
}
```

Now the keyboard disappears as soon as you flip the device.

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, although it gets temporarily hidden under the `LandscapeViewController`. When you rotate back from landscape, the pop-up is still there.

I find that a little strange. You may have been doing all sorts of stuff on the landscape screen and you'll probably have forgotten that the Detail pop-up was still open. Upon returning to the table view, you're suddenly confronted with it. I think it would be better if the app dismissed the pop-up before rotating.

Exercise. See if you can fix that one.

Answer: Again you already have all the code for this. `DetailViewController` has a public method named `dismissFromParentViewController`, so you can simply call that. The only problem is that you don't have a pointer to the `DetailViewController` (only as a local variable in `didSelectRowAtIndexPath`) so you cannot simply do `[_detailViewController dismissFromParentViewController]`. Unless you make an instance variable, of course.

➤ Add the instance variable to **SearchViewController.m**:

```
DetailViewController *_detailViewController;
```

➤ Add a new line to the bottom of `showLandscapeViewWithDuration:`

```
- (void)showLandscapeViewWithDuration:(NSTimeInterval)duration
{
  if (_landscapeViewController == nil) {
    . . .

    [_detailViewController dismissFromParentViewController];
  }
}
```

Naturally, you have to put the new `DetailViewController` object into this variable when you create it. That happens in the `didSelectRowAtIndexPath` method.

➤ Change `didSelectRowAtIndexPath` to:

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  . . .

  _detailViewController = controller;
}
```

The new line puts the value from the local variable `controller` into the `_detailViewController` instance variable.

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

I notice one small problem, the `NSLog()` in `DetailViewController`'s `dealloc` method no longer gets triggered when the pop-up is dismissed. That means the app is not deallocating this object anymore: you have a memory leak!

Exercise. Explain why this happens. □

Answer: By creating the `_detailViewController` instance variable, you have added a new strong pointer to that object. This strong pointer never gets set to `nil` so it keeps the `DetailViewController` object alive, at least until you open a new pop-up.

Try it: When you tap another row, you should see the `dealloc` message for the previous `DetailViewController`. Because you put a new value into `_detailViewController`, there is no more strong pointer to the old `DetailViewController` object and it can finally go away.

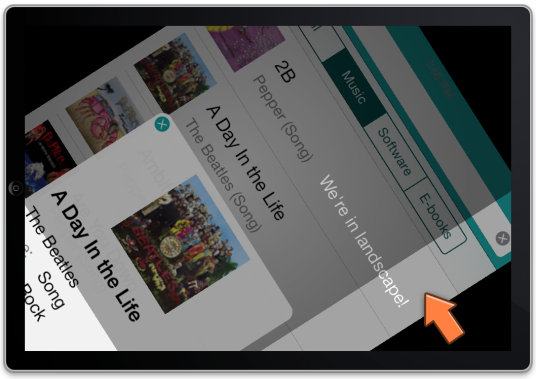
There are two ways to fix this, you can set `_detailViewController` explicitly to `nil` in `hideLandscapeViewWithDuration:` after dismissing it, or you can make the instance variable weak instead of strong. Let's use a weak pointer, that's what they are there for.

➤ Change the declaration of the instance variable to:

```
__weak DetailViewController *_detailViewController;
```

Now this pointer is no longer keeping the object alive and as soon as the object is deallocated, `_detailViewController` automatically becomes `nil`. Try it out; you should see the `NSLog()` again.

If you look really carefully while the screen rotates, you can see a glitch at the right side of the screen. Remember how the search and navigation bars didn't stretch to fill up the extra space? The same thing happens to the gradient view:



Because you didn't create `GradientView` in Interface Builder you cannot easily set constraints on it, like you did for the search bar and navigation bar. Instead, you can change its **autosizing** behavior from code.

Autosizing is what iOS developers used before Auto Layout existed. Using the `autoresizingMask` property you can tell a view what it should do when its superview changes in size. You have a variety of options, such as: do nothing, stick to a certain side, or change in size proportionally. The possibilities are much more limited than what you can do with Auto Layout, but for many scenarios autosizing is good enough.

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

➤ Change **GradientView.m**'s `initWithFrame:` method to:

```
- (id)initWithFrame:(CGRect)frame
{
  if ((self = [super initWithFrame:frame])) {
    self.backgroundColor = [UIColor clearColor];
    self.autoresizingMask = UIViewAutoresizingFlexibleWidth |
                            UIViewAutoresizingFlexibleHeight;
```

```
  }
  return self;
}
```

This tells the view that it should change both its width and its height proportionally when the superview it belongs to changes its size (due to being rotated or otherwise). In practice this means the `GradientView` will always cover the same area that its superview covers and there should be no more gaps, even if the device gets rotated.

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

Enums

When setting the `autoresizingMask` property, you combined two values, `UIViewAutoresizingFlexibleWidth` and `UIViewAutoresizingFlexibleHeight`. The notation for that is the `|` pipe symbol. This is also known as a **bitwise OR** (as opposed to a "logical OR", which uses two `||`'s) but to keep it simple you can think of this as adding two numbers together.

The symbols `UIViewAutoresizingFlexibleWidth` and `UIViewAutoresizingFlexibleHeight` are part of what is known as an **enum**. The word enum is short for enumeration and that probably still doesn't tell you much, but it's a special datatype from the C language that lets you define a range of symbolic names. Instead of using the numeric values 1, 2, 3, 4 and so on you use an `enum` to give these numbers symbolic names.

For the fun of it, let's take a look at the actual `enum` for the autoresizing mask. In Xcode, place the text cursor on the word `UIViewAutoresizingFlexibleWidth` and press **Ctrl-Cmd-J** (or right click and choose the **Jump to Definition** command from the menu). This will open the file **UIView.h** from the UIKit framework in the source code editor.

You're now looking at a piece of the actual UIKit source code. You can do this for all the .h files from all the frameworks and it's a great way to learn more about these frameworks. Header files can really be considered part of the official documentation and they often contain enlightening comments. They're worth looking into! (Unfortunately, we don't get to see the .m files from these frameworks, only the headers.)

The enum for `UIViewAutoresizing` looks as follows:

```
typedef NS_OPTIONS(NSUInteger, UIViewAutoresizing) {
    UIViewAutoresizingNone                 = 0,
    UIViewAutoresizingFlexibleLeftMargin   = 1 << 0,
    UIViewAutoresizingFlexibleWidth        = 1 << 1,
    UIViewAutoresizingFlexibleRightMargin  = 1 << 2,
    UIViewAutoresizingFlexibleTopMargin    = 1 << 3,
```

```
    UIViewAutoresizingFlexibleHeight       = 1 << 4,
    UIViewAutoresizingFlexibleBottomMargin = 1 << 5
};
```

The `NS_OPTIONS` (and `NS_ENUM`) macros are a special notation for defining enums. This is equivalent to writing:

```
typedef enum {

  . . .

} UIViewAutoresizing;
```

The `UIViewAutoresizing` enum defines seven symbolic names that can be used with the `autoresizingMask` property. The notation "1 << 0" may be a little freaky but what this really means is:

```
typedef NS_OPTIONS(NSUInteger, UIViewAutoresizing) {
    UIViewAutoresizingNone                 = 0,
    UIViewAutoresizingFlexibleLeftMargin   = 1,
    UIViewAutoresizingFlexibleWidth        = 2,
    UIViewAutoresizingFlexibleRightMargin  = 4,
    UIViewAutoresizingFlexibleTopMargin    = 8,
    UIViewAutoresizingFlexibleHeight       = 16,
    UIViewAutoresizingFlexibleBottomMargin = 32
};
```

This gives a unique numeric value to each of these symbols, which lets you add them up to make combinations. You could have written the code earlier as,

```
self.autoresizingMask = 18;  // FlexibleWidth = 2 +
                             // FlexibleHeight = 16
```

but that wouldn't have been half as clear. Using a symbolic name is much more readable than a "naked" number.

UIView.h has a few other enums, such as this one:

```
typedef NS_ENUM(NSInteger, UIViewAnimationCurve) {
    UIViewAnimationCurveEaseInOut,  // slow at beginning and end
    UIViewAnimationCurveEaseIn,     // slow at beginning
    UIViewAnimationCurveEaseOut,    // slow at end
    UIViewAnimationCurveLinear
};
```

This enum does not have any numbers behind the symbols, which means they get default values starting from 0:

```
typedef NS_ENUM(NSInteger, UIViewAnimationCurve) {
    UIViewAnimationCurveEaseInOut = 0,
    UIViewAnimationCurveEaseIn    = 1,
```

```
    UIViewAnimationCurveEaseOut   = 2,
    UIViewAnimationCurveLinear    = 3
};
```

Notice that this enum has a name, `UIViewAnimationCurve`. An enum with a name can act as a datatype, so you can use this to make the compiler force you to always use one of these symbolic names. For example, if you define a variable like this,

```
UIViewAnimationCurve curve;
```

then the compiler will complain if you try to put any value in that variable that is not one of the symbols from the corresponding enum. Enum types are really handy and are used all over Apple's frameworks. For example, `UIStatusBarStyle` is an enum and so is `UIInterfaceOrientation`.

A better "hide" animation

Something else bothers me: the animation of the pop-up falling down the screen is a little weird in combination with the rotation animation. There's too much happening on the screen at once, to my taste. Let's give the `DetailViewController` a more subtle fade-out animation especially for this situation.

When you tap the X button to dismiss the pop-up, you'll still make it fall down 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 could give `DetailViewController` a second "dismiss" method, something like `dismissFromParentViewControllerWithFadeOut`. That will work, but instead let's add a parameter to the existing method that specifies how it will animate the pop-up's dismissal. You can use an enum for this.

➤ Add the following to **DetailViewController.h**, above the `@interface` line:

```
typedef NS_ENUM(NSUInteger, DetailViewControllerAnimationType) {
  DetailViewControllerAnimationTypeSlide,
  DetailViewControllerAnimationTypeFade
};
```

This defines a new enum named `DetailViewControllerAnimationType`. It has two values, `DetailViewControllerAnimationTypeSlide` and `...AnimationTypeFade`. It's customary in Objective-C to be verbose so enums tend to have long names. But you should be used to long names by now.

➤ In the `@interface` for `DetailViewController`, replace the signature of the existing dismiss method with:

```
- (void)dismissFromParentViewControllerWithAnimationType:
                (DetailViewControllerAnimationType)animationType;
```

The method now has a parameter named `animationType` that can only be one of the values from the enum you just defined.

› Replace the implementation of the dismiss method with:

```
- (void)dismissFromParentViewControllerWithAnimationType:
                (DetailViewControllerAnimationType)animationType
{
  [self willMoveToParentViewController:nil];

  [UIView animateWithDuration:0.4 animations:^{

    if (animationType ==
                    DetailViewControllerAnimationTypeSlide) {
      CGRect rect = self.view.bounds;
      rect.origin.y += rect.size.height;
      self.view.frame = rect;
    } else {
      self.view.alpha = 0.0f;
    }
    _gradientView.alpha = 0.0f;

  } completion:^(BOOL finished) {
    [self.view removeFromSuperview];
    [self removeFromParentViewController];
    [_gradientView removeFromSuperview];
  }];
}
```

It's mostly the same as before, except that now it checks the value of the `animationType` parameter. If it's "slide", you do the same as before, but if it's "fade" you simply set the view's alpha value to 0 in order to fade it out.

› Change the `close:` method to:

```
- (IBAction)close:(id)sender
{
  [self dismissFromParentViewControllerWithAnimationType:
                        DetailViewControllerAnimationTypeSlide];
}
```

› In **SearchViewController.m**'s `showLandscapeViewWithDuration:` method, change the line to:

```
[_detailViewController
      dismissFromParentViewControllerWithAnimationType:
                          DetailViewControllerAnimationTypeFade];
```

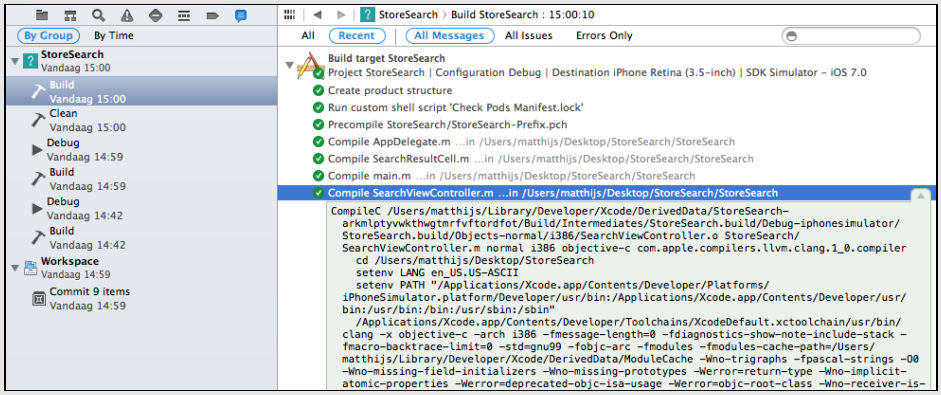
And that does it. If you wanted to create more animations that can be used on dismissal, you only have to add a symbol to the enum and check for it in the dismiss method.

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.

You can find the project files for the app up to this point under **08 - Landscape** in the tutorial's Source Code folder.

The build log

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



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

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

Adding the scroll view

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

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

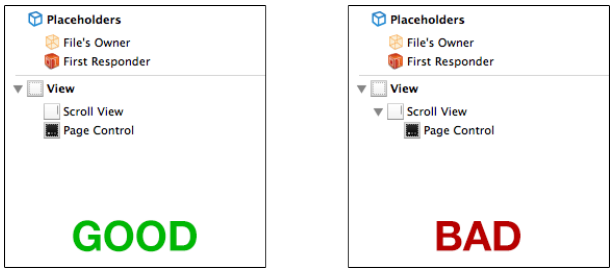
In this section you're going to use a scroll view of your own, in combination with a **paging control**, so you can show the artwork for all the search results even if there are more images than fit on the screen at once.

› Open **LandscapeViewController.xib** and delete the label.

› Now drag a new **Scroll View** into the nib. Make it as big as the screen (568 points wide, 320 points high).

› Drag a new **Page Control** object into the nib. This gives you a small view with three white dots. Make it 568 points wide so that it spans the entire screen. Set its position to X: 0, Y: 284.

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



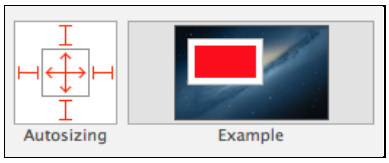
The Page Control should be a "sibling" of the Scroll View, not a child

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

You're designing the nib in 4-inch mode. But it should also work OK for the smaller 3.5-inch screen. With Auto Layout you can pin views to the sides of the window so that they resize when the window resizes, but you can't do that here because you disabled Auto Layout for this nib. The solution is to use autosizing, also known as **springs and struts**.

You already saw that you could set a view's autosizing mask from code. Of course you can also do this from Interface Builder.

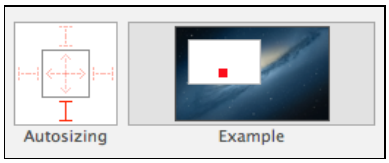
› Select the **Scroll View** and go to the **Size inspector**. Its **Autosizing** options should look like this:



Autosizing options for the scroll view

This means the scroll view will always be as big as its superview. If you hover the mouse over the Example box, it shows an animation of how the view will behave when its superview resizes. The red box represents the view; the white box the superview.

▸ Select the **Page Control**. Change its **Autosizing** options to:



By deselecting everything except the T-bar at the bottom, you state that this view should always be centered horizontally. In the vertical direction it should stay pinned to the bottom edge of the superview.

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

The final design of the landscape nib

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

▸ Add the outlets to the class extension in **LandscapeViewController.m**, and connect them in Interface Builder:

```
@interface LandscapeViewController ()
@property (nonatomic, weak) IBOutlet UIScrollView *scrollView;
@property (nonatomic, weak) IBOutlet UIPageControl *pageControl;
@end
```

If you run the app and flip to landscape, nothing much happens yet. The screen is black with a few dots at the bottom. For the scroll view to do anything you have to add some content to it.

▸ Replace the `viewDidLoad` method with:

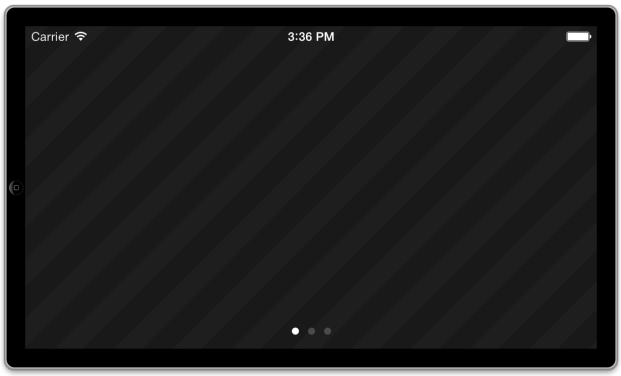
```
- (void)viewDidLoad
{
  [super viewDidLoad];

  self.scrollView.backgroundColor =
          [UIColor colorWithPatternImage:
                   [UIImage imageNamed:@"LandscapeBackground"]];

  self.scrollView.contentSize = CGSizeMake(
                      1000, self.scrollView.bounds.size.height);
}
```

Two things happen here: first you put an image on the scroll view's background so you can actually see something happening when you scroll through it.

An image? But you're setting the `backgroundColor` property, which is a `UIColor`, not a `UIImage`?! Yup, that's true, but `UIColor` has a cool trick that lets you use a tile-able image for a color. If you take a peek at the **LandscapeBackground.png** image in the asset catalog you'll see that it is a small square. By setting this image as a pattern image on the background, you get a repeatable image that fills the whole screen. You can use tile-able images anywhere you can use a `UIColor`.

The second thing, and this is very important when dealing with scroll views, is that you set the `contentSize` property. This will tell the scroll view how big its insides are. You don't change the frame (or bounds) of the scroll view if you want its insides to be bigger, you set the `contentSize` property instead. People often forget this step and then they wonder why their scroll view doesn't scroll. Unfortunately, you cannot do this from Interface Builder, so it must be done from within the 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 when scrolling then you placed the Page Control inside the Scroll View. Open the nib and in the outline pane drag the Page Control on top of the Scroll View instead.)

Adding buttons for the search results

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

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

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

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

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

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

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

```
@property (nonatomic, strong) NSArray *searchResults;
```

You set this property in `SearchViewController` upon rotation to landscape. You have to be sure to do that before you access the `LandscapeViewController`'s `view` property, because that will trigger the view to be loaded. The view controller needs to read from the `searchResults` array to build up the contents of its scroll view, but if you access `_landscapeViewController.view` before setting `searchResults` then this property will still be `nil`. The order in which you do things matters!

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

```
- (void)showLandscapeViewWithDuration:(NSTimeInterval)duration
{
  if (_landscapeViewController == nil) {
    _landscapeViewController = [[LandscapeViewController alloc]
         initWithNibName:@"LandscapeViewController" bundle:nil];

    _landscapeViewController.searchResults = _searchResults;

    . . .
```

Let's go make those buttons.

➤ First, import the `SearchResult` object in **LandscapeViewController.m** as you will be needing it:

```
#import "SearchResult.h"
```

➤ Remove the line that sets `self.scrollView.contentSize` from `viewDidLoad`.

➤ Add a new instance variable:

```
@implementation LandscapeViewController
{
  BOOL _firstTime;
}
```

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

➤ Change the init method to do that:

```
- (id)initWithNibName:(NSString *)nibNameOrNil
               bundle:(NSBundle *)nibBundleOrNil
{
  self = [super initWithNibName:nibNameOrNil
                         bundle:nibBundleOrNil];
  if (self) {
    _firstTime = YES;
  }
  return self;
}
```

➤ Add a new method, `viewWillLayoutSubviews`, below `viewDidLoad`:

```
- (void)viewWillLayoutSubviews
{
  [super viewWillLayoutSubviews];

  if (_firstTime) {
    _firstTime = NO;

    [self tileButtons];
  }
}
```

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

You may think that `viewDidLoad` is a good place for that, but at the point in the view controller's lifecycle when `viewDidLoad` is called, the view is not added into the view hierarchy yet. As a result it doesn't know how large it should be. Only after `viewDidLoad` is done does the view get resized to fit on the 3.5- or 4-inch screen.

The only safe place to perform calculations based on the final size of the view (i.e. any calculations that use the view's `frame` or `bounds`) is in `viewWillLayoutSubviews`. However, that method may be invoked more than once; for example, when the landscape view gets removed from the screen again. You use the `_firstTime` variable to make sure you only place the buttons once.

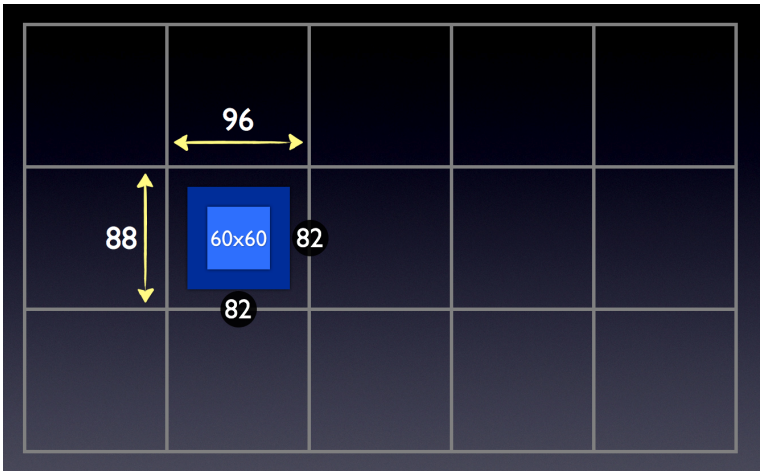
▸ Add the `tileButtons` method below `viewWillLayoutSubviews`. This is a big one!

```
- (void)tileButtons
{
  int columnsPerPage = 5;
  CGFloat itemWidth = 96.0f;
  CGFloat x = 0.0f;
  CGFloat extraSpace = 0.0f;

  CGFloat scrollViewWidth = self.scrollView.bounds.size.width;
  if (scrollViewWidth > 480.0f) {
    columnsPerPage = 6;
    itemWidth = 94.0f;
    x = 2.0f;
    extraSpace = 4.0f;
  }

  const CGFloat itemHeight = 88.0f;
  const CGFloat buttonWidth = 82.0f;
  const CGFloat buttonHeight = 82.0f;
  const CGFloat marginHorz = (itemWidth - buttonWidth)/2.0f;
  const CGFloat marginVert = (itemHeight - buttonHeight)/2.0f;

  int index = 0;
  int row = 0;
  int column = 0;

  for (SearchResult *searchResult in self.searchResults) {

    UIButton *button = [UIButton
                            buttonWithType:UIButtonTypeSystem];

    button.backgroundColor = [UIColor whiteColor];
    [button setTitle:[NSString stringWithFormat:@"%d", index]
            forState:UIControlStateNormal];
```

```
      button.frame = CGRectMake(x + marginHorz, 20.0f +
          row*itemHeight + marginVert, buttonWidth, buttonHeight);

      [self.scrollView addSubview:button];

      index++;
      row++;
      if (row == 3) {
        row = 0;
        column++;
        x += itemWidth;

        if (column == columnsPerPage) {
          column = 0;
          x += extraSpace;
        }
      }
    }

    int tilesPerPage = columnsPerPage * 3;
    int numPages = ceilf([self.searchResults count] /
                                            (float)tilesPerPage);
    self.scrollView.contentSize = CGSizeMake(
                               numPages*scrollViewWidth,
                               self.scrollView.bounds.size.height);

    NSLog(@"Number of pages: %d", numPages);
  }
```

There are a few different things going on here. First, the method must decide on how big the buttons will be. You already determined that each search result gets a rectangle of 96 by 88 points (or 94×88 on 4-inch devices) 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, so that leaves a small margin between each button and its neighbors.



The dimensions of the buttons in the 5x3 grid

That margin calculation is the first thing you do:

```
int columnsPerPage = 5;
CGFloat itemWidth = 96.0f;
CGFloat x = 0.0f;
CGFloat extraSpace = 0.0f;

CGFloat scrollViewWidth = self.scrollView.bounds.size.width;
if (scrollViewWidth > 480.0f) {
  columnsPerPage = 6;
  itemWidth = 94.0f;
  x = 2.0f;
  extraSpace = 4.0f;
}

const CGFloat itemHeight = 88.0f;
const CGFloat buttonWidth = 82.0f;
const CGFloat buttonHeight = 82.0f;
const CGFloat marginHorz = (itemWidth - buttonWidth)/2.0f;
const CGFloat marginVert = (itemHeight - buttonHeight)/2.0f;
```

The `columnsPerPage` variable keeps track of how many columns will fit in a single screenful of buttons. The number of columns on a 4-inch screen is 6 but 568 doesn't evenly divide by 6, so the `extraSpace` variable is used to adjust for the 4 points that are left over.

None of the other values – `itemHeight`, `buttonWidth`, and so on – change during the duration of the app so you can make them `const`. I could have calculated the values of `marginHorz` and `marginVert` by hand and put the actual value in the code, but you might as well make the compiler do that for you. After all, calculating things is what computers are good at.

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

```
      UIButton *button = [UIButton
                              buttonWithType:UIButtonTypeSystem];

      button.backgroundColor = [UIColor whiteColor];
      [button setTitle:[NSString stringWithFormat:@"%d", index]
              forState:UIControlStateNormal];

      button.frame = CGRectMake(x + marginHorz, 20.0f +
          row*itemHeight + marginVert, buttonWidth, buttonHeight);

      [self.scrollView addSubview:button];
```

When you make a button by hand you always have to set its frame. Using the stuff you figured out earlier you can determine the position and size of the button.

For debugging purposes you give each button a title with an index, counting from 0. 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.

You add the new button object as a subview to the `UIScrollView`. After the first 15 buttons (or 18 on 4-inch) this places any subsequent button out of the visible range of the scroll view, but that's the whole point. As long as you set the `contentSize` accordingly, the user can scroll to get to those other buttons.

```
      index++;
      row++;
      if (row == 3) {
        row = 0;
        column++;
        x += itemWidth;

        if (column == columnsPerPage) {
          column = 0;
          x += extraSpace;
        }
```

You use the `x` and `row` variables to position the buttons. You go from top to bottom (by increasing `row`); when you've reached the bottom (row 3) you go up again 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 the left-over space to `x`. This only has an effect on 4-inch screens because for 3.5-inch screens `extraSpace` is 0.

```
    int tilesPerPage = columnsPerPage * 3;
```

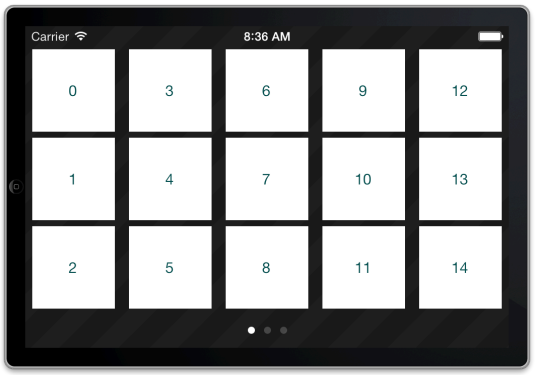
```
    int numPages = ceilf([self.searchResults count] /
                                            (float)tilesPerPage);
    self.scrollView.contentSize = CGSizeMake(
                              numPages*scrollViewWidth,
                              self.scrollView.bounds.size.height);
```

At the end of the method you calculate the `contentSize`. We want the user to be able to "page" through these results, rather than simply scroll (a feature that you'll enable shortly) so you should always make the content width a multiple of 480 or 568 points (the width of a single page). With a simple formula you can determine how many pages you need.

> **Note:** Remember that dividing an integer value by an integer always results in an integer, so you cannot simply divide by `tilesPerPage`. You need to promote the datatype of this variable to `float` in order to preserve what is behind the decimal point. You do that with the cast operator, `(float)`. Then you always round up using the `ceilf()` function because even if there is a remainder of just one search result you have to give that a new page.

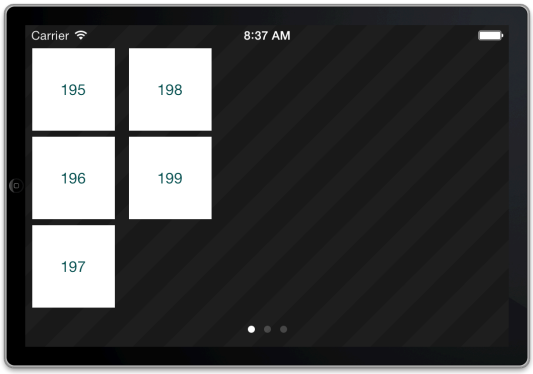
I also threw in an `NSLog()` for good measure, so you can verify that you really end up with the right amount of pages.

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



The landscape view has buttons

The last page looks like this:



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 you should test a few different scenarios. What happens when there are fewer results than 15? What happens when there are exactly 15 results? How about 16, one more than fit on a single page? The easiest way to create this situation is to change the `&limit` parameter in the search URL.

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

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

Paging

So far the Page Control at the bottom of the screen has always shown three dots. And there wasn't much paging to be done on the scroll view either. In case you're wondering what "paging" means: if the user has dragged the scroll view a certain amount, the scroll view should snap to a new page.

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

▸ Open the **LandscapeViewController** nib and check the **Paging Enabled** option for the scroll view. There, that was easy. Now run the app and the scroll view will let you page rather than scroll.

That's cool but you also need to do something with the page control at the bottom.

▸ Add this line to `viewDidLoad`:

```
self.pageControl.numberOfPages = 0;
```

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

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

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

This sets the number of dots that the page control displays to the number of pages that you calculated. However, the active dot (the white one) still isn't synchronized with the active page in the scroll view; it never changes unless you tap in the page control and even then it has no effect on the scroll view. You have to make the page control talk to the scroll view, and vice versa.

➤ Change the `@interface` line to:

```
@interface LandscapeViewController () <UIScrollViewDelegate>
```

You're going to make the view controller the delegate of the scroll view so it will be notified when the user is flicking through the pages.

➤ In Interface Builder, **Ctrl-drag** from the Scroll View to File's Owner and select **delegate**.

➤ Add the following method to **LandscapeViewController.m**:

```
- (void)scrollViewDidScroll:(UIScrollView *)theScrollView
{
  CGFloat width = self.scrollView.bounds.size.width;
  int currentPage =
         (self.scrollView.contentOffset.x + width/2.0f) / width;
  self.pageControl.currentPage = currentPage;
}
```

This is one of the `UIScrollViewDelegate` methods. You figure out what the index of the current page is by looking at the `contentOffset` property of the scroll view. This property determines how far the scroll view has been scrolled and is updated while you're dragging the scroll view. If the content offset gets beyond halfway on the page, the scroll view will flick to the next page. In that case, you update the `pageControl`'s active page number. Unfortunately, the scroll view doesn't simply tell us, "The user has flipped to page X" so you have to calculate this yourself.

You also need to know when the user taps on the Page Control. There is no delegate for this but you can use a regular action method.

➤ Add the action method:

```
- (IBAction)pageChanged:(UIPageControl *)sender
{
  self.scrollView.contentOffset = CGPointMake(
     self.scrollView.bounds.size.width * sender.currentPage, 0);
```

```
}
```

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

› In Interface Builder, **Ctrl-drag** from the Page Control to File's Owner and select **pageChanged:**.

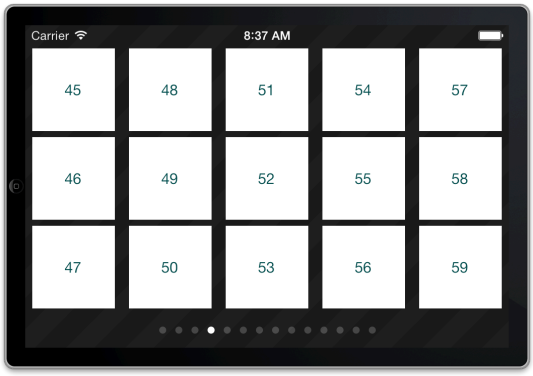
› Try it out, the page control and the scroll view should now be in sync. The transition from one page to another after tapping in the page control is still a little abrupt, though. An animation would help here.

Exercise. See if you can animate what happens in `pageChanged:`. □

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

```
- (IBAction)pageChanged:(UIPageControl *)sender
{
  [UIView animateWithDuration:0.3
    delay:0
    options:UIViewAnimationOptionCurveEaseInOut
    animations:^{
      self.scrollView.contentOffset = CGPointMake(
        self.scrollView.bounds.size.width * sender.currentPage,
        0);
    }
    completion:nil];
}
```

I used a version of the `UIView` animation method that allows us to specify options because the "Ease In, Ease Out" timing (`UIViewAnimationOptionCurveEaseInOut`) looks good here.



We've got paging!

› This is a good time to commit.

Putting artwork on the buttons

First let's give the buttons a nicer look:

▸ Replace the button creation code in `tileButtons` with:

```
  UIButton *button = [UIButton buttonWithType:UIButtonTypeCustom];

[button setBackgroundImage:
                 [UIImage imageNamed:@"LandscapeButton"]
        forState:UIControlStateNormal];

button.frame = . . .
```

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

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

The buttons now have a custom background image

Now you will have to load the artwork images (if they haven't been already downloaded and cached yet by the table view) and put them on the button. Problem: You're dealing with `UIButtons` here, not `UIImageViews`, so you cannot simply use that category from AFNetworking. Fortunately, they also provided a similar category for `UIButton`!

▸ Import this category into **LandscapeViewController.m**:

```
#import <AFNetworking/UIButton+AFNetworking.h>
```

▸ And add a new method:

```
- (void)downloadImageForSearchResult:
                                  (SearchResult *)searchResult
        andPlaceOnButton:(UIButton *)button
{
  NSURL *url = [NSURL URLWithString:searchResult.artworkURL60];
```

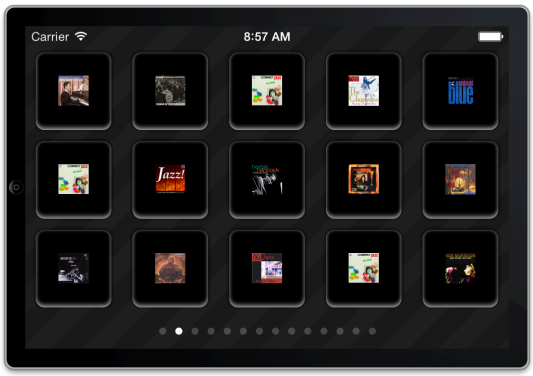
```
    [button setImageForState:UIControlStateNormal withURL:url];
}
```

First you create an `NSURL` object that contains the URL to the 60×60 pixel artwork and then you use the `setImageForState:withURL:` method to put that image on the button. That is not a standard `UIButton` method, but comes from the AFNetworking+UIButton category.

▸ Add the following line to `tileButtons` to call this new method, right after where you create the button:

```
[self downloadImageForSearchResult:searchResult
                 andPlaceOnButton:button];
```

And that should do it. If you run the app, you'll get some cool-looking buttons:



Showing the artwork on the buttons

Of course, there is always some small problem. Literally. The images are 60×60 pixels, which on a Retina screen scales down to really small images. The current version of the iTunes web service does not include Retina-sized images. It would be better if the app scaled these images up to fill the button; they will look more blocky that way but at least you can see them better.

So how do you do that? The AFNetworking method does not have a "scale" parameter that lets you scale the images up or down. In fact, it doesn't give you access to the image at all. However, there is another version of that method that provides more options:

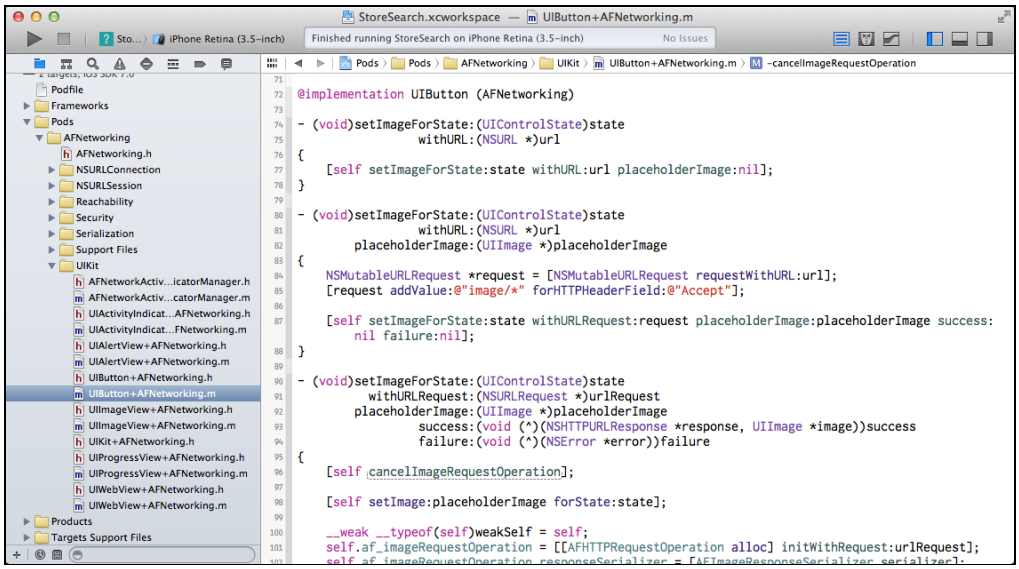
```
setImageForState:withURLRequest:placeholderImage:success:failure:
```

This method takes a few extra parameters, including a success block. That's what we're interested in because it gives you the image. And once you have a reference to the `UIImage` object you can scale it up or down.

Unlike the other method, this one takes an `NSURLRequest` object instead of an `NSURL`. That makes it more versatile, but I wonder how to make that `NSURLRequest` object.

How to find out about that? Well, you do have the full source code to AFNetworking in the Pods project, so let's dig into their code to see what they are doing.

➤ Go into the **Pods** project and open the **Pods** group. Inside is a folder for **AFNetworking** with a subfolder **UIKit** that contains the source file **UIButton+AFNetworking.m**. Open that file to take a peek.



The AFNetworking source code

You typically wouldn't want to change anything here, but it's fine to look around to try to make sense of it. Studying other people's source code is a great way to learn.

You can see that the method you're currently using, `setImageForState:withURL:` doesn't do much. It just calls through to the method below it, which has an additional `placeholderImage` parameter.

Ah ha! Inside that method you can see how the `NSURLRequest` object is created (it's actually an `NSMutableURLRequest`, a subclass of `NSURLRequest`):

```
NSMutableURLRequest *request = [NSMutableURLRequest
                                        requestWithURL:url];
[request addValue:@"image/*" forHTTPHeaderField:@"Accept"];
```

That's important to remember, because you'll need to do the same thing. After it creates this object, the method calls the big `setImageForState` method and passes it the `NSURLRequest`. Great, we've seen enough now to be able to improve the image loading code.

➤ Go back to **LandscapeViewController.m** and change the `downloadImageForSearchResult:andPlaceOnButton:` method (don't you love these long names?):

```
- (void)downloadImageForSearchResult:
                                  (SearchResult *)searchResult
        andPlaceOnButton:(UIButton *)button
{
  NSURL *url = [NSURL URLWithString:searchResult.artworkURL60];

  // 1
  NSMutableURLRequest *request = [NSMutableURLRequest
                                              requestWithURL:url];
  [request addValue:@"image/*" forHTTPHeaderField:@"Accept"];

  // 2
  __weak UIButton *weakButton = button;

  // 3
  [button setImageForState:UIControlStateNormal
        withURLRequest:request placeholderImage:nil
        success:^(NSHTTPURLResponse *response, UIImage *image) {

    // 4
    UIImage *unscaledImage = [UIImage
                imageWithCGImage:image.CGImage scale:1.0
                orientation:image.imageOrientation];

    [weakButton setImage:unscaledImage
                forState:UIControlStateNormal];
  }
  failure:nil];
}
```

Here is what this does, step-by-step:

1. Create the `NSMutableURLRequest`. I simply copied this from the AFNetworking source code.
2. The `setImageForState` method takes a block of code that is performed when the image is successfully downloaded. Inside that block you place the image on the button, which means that the block **captures** the `button` variable. That creates an **ownership cycle**, because the button owns the block and the block owns the button, resulting in a possible memory leak.

   To prevent this, you create a new variable `weakButton` that refers to the same button but is a weak pointer. The button still owns the block but now the block doesn't own the button back. Crisis averted.
3. Call the big `setImageForState` method to download the image.
4. Inside the success block you need to do two things: scale the image and place the image on the button. In this case, "scaling" doesn't mean that you create a

new image that is twice as big. By passing 1.0 to the `scale` parameter you simply tell `UIImage` that it should not treat this as a Retina image.

It's a bit more work than before, but not too bad – and now you have full control over how the images appear.

- Try it out. The images should be a lot bigger on the Retina screen:

The images are no longer scaled down on Retina

It's always a good idea to clean up after yourself, also in programming. Imagine this: what would happen if the app is still downloading images and the user flips the device back to portrait mode? The `LandscapeViewController` is deallocated but the image downloads keep going. That is exactly the sort of situation that can crash your app if you don't handle it properly.

> **Note:** Here it won't do any harm, actually. Whenever an image successfully downloads, the success block tries to call the `setImage:forState:` method on `weakButton`. Because that is a weak pointer, and the buttons are all deallocated at this point, `weakButton` now points to `nil`. Sending messages to `nil` is allowed in Objective-C, so that won't cause trouble (in other languages it *will* crash). However, to conserve resources the app should really stop downloading these images because they end up nowhere. Otherwise it's just wasting bandwidth and battery life, and users don't take too kindly to apps that do.

- Tell the `dealloc` method to cancel any operations that are still on the way:

```
- (void)dealloc
{
  NSLog(@"dealloc %@", self);

  for (UIButton *button in self.scrollView.subviews) {
    [button cancelImageRequestOperation];
  }
}
```

This will stop the download for any button whose image was still pending or in transit.

Exercise. Despite what the iTunes web service promises, not all of the artwork is truly 60×60 pixels. Some of it is bigger, some is not even square, and so it might not always fit nicely in the button.

Using the image sizing code from MyLocations, always resize the image to 60×60 points before you put it on the button. Note that we're talking points here, not pixels, so on Retina devices the image should actually end up being 120×120 pixels big. □

➤ Commit your changes.

You can find the project files for the app up to this point under **09 - Landscape with Artwork** in the tutorial's Source Code folder.

> **Note:** In this section you learned how to create a grid-like view using a `UIScrollView`. iOS comes with a versatile class, `UICollectionView`, that lets you do the same thing – and much more! – without having to resort to the sort of math that you did in `tileButtons`. To learn more about `UICollectionView`, check out the book *iOS 6 by Tutorials*: http://www.raywenderlich.com/store

Refactoring the search

If you start a search and switch to landscape while the results are still downloading, then the landscape view will remain empty. It would be nice to also show an activity spinner in landscape while the search is taking place. You can reproduce this situation by artificially slowing down your network connection using the SpeedLimit preferences pane or the Network Link Conditioner tool.

So how can `LandscapeViewController` tell what state the search is in? Its `searchResults` property will either be `nil` if no search was done yet, or have an `NSArray` object with zero or more `SearchResult` objects. But just by looking at the array object it cannot determine whether the search is still going, or whether the search is done and has no results. In both cases, the `searchResults` array will have a count of 0.

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

➤ If you want, you can create a new branch for this in Git. This is a pretty invasive change in the code and there is always a risk that it doesn't actually work as well as you hoped. By making the changes in a new branch, you can commit every once in a while without messing up the master branch.

➤ Create a new file, **Objective-C class** template. Name it **Search**, subclass of **NSObject**.

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

```
@interface Search : NSObject

@property (nonatomic, assign) BOOL isLoading;
@property (nonatomic, readonly, strong) NSMutableArray
                                              *searchResults;

- (void)performSearchForText:(NSString *)text
                    category:(NSInteger)category;

@end
```

You've given this class two properties 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.

➤ Add a new class extension inside **Search.m**:

```
@interface Search ()
@property (nonatomic, readwrite, strong) NSMutableArray
                                               *searchResults;
@end
```

In **Search.h** you also defined this `searchResults` property, but there it said "readonly". That means users of this class can only read what is inside the `searchResults` array but they cannot replace the `searchResults` array with another one (why would they?).

However, inside the `Search` class you do want full control over the `searchResults` array and therefore you re-declare it as "readwrite" in the class extension. That's a little trick to make a property read-only to other objects but fully accessible to your own class.

➤ Add the following two methods to the implementation:

```
- (void)dealloc
{
  NSLog(@"dealloc %@", self);
}
```

```
- (void)performSearchForText:(NSString *)text
                    category:(NSInteger)category
{
  NSLog(@"Searching...");
}
```

The `performSearchForText:category:` method doesn't do much yet but that's OK. First I want to put this `Search` object into `SearchViewController` and when that compiles, you will move all the logic over. Small steps!

Just to make sure there aren't any memory leaks, I put an `NSLog()` in `dealloc`. Whenever you do a new search, the old `Search` object is no longer needed and should be deallocated.

Let's make the changes to **SearchViewController.m**. 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.

➤ As always, you need to import the objects you're going to use:

```
#import "Search.h"
```

➤ Remove the declarations for the following instance variables,

```
  NSMutableArray *_searchResults;
  BOOL _isLoading;
  NSOperationQueue *_queue;
```

and replace them with this one:

```
  Search *_search;
```

You can remove a lot of code from this view controller.

➤ Remove the `initWithNibName` method.

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

- `urlWithSearchText:category:`
- `parseDictionary:`
- `parseTrack:`
- `parseAudioBook:`
- `parseSoftware:`
- `parseEBook:`

➤ Replace the `performSearch` method with:

```
- (void)performSearch
{
  _search = [[Search alloc] init];
  NSLog(@"allocated %@", _search);

  [_search performSearchForText:self.searchBar.text
           category:self.segmentedControl.selectedSegmentIndex];

  [self.tableView reloadData];
  [self.searchBar resignFirstResponder];
}
```

This allocates a new `Search` object and lets that do all the work. Of course it still reloads the table view (to show the activity spinner) and hides the keyboard.

There are a few places in the code that still use the `_searchResults` array even though that no longer is an instance variable. You should change them to use the `searchResults` property from the `Search` object instead. Likewise for `isLoading`.

➤ Change `numberOfRowsInSection` to:

```
- (NSInteger)tableView:(UITableView *)tableView
              numberOfRowsInSection:(NSInteger)section
{
  if (_search == nil) {
    return 0;  // Not searched yet
  } else if (_search.isLoading) {
    return 1;  // Loading...
  } else if ([_search.searchResults count] == 0) {
    return 1;  // Nothing Found
  } else {
    return [_search.searchResults count];
  }
}
```

➤ In `showLandscapeViewWithDuration:`, change the line that sets the `searchResults` property to:

```
_landscapeViewController.search = _search;
```

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

➤ In `segmentChanged:` you now have to check for the existence of the `_search` variable rather than `_searchResults`:

```
- (IBAction)segmentChanged:(UISegmentedControl *)sender
{
```

```
    if (_search != nil) {
      [self performSearch];
    }
  }
```

➤ Anywhere else in the code that it says `_isLoading` or `_searchResults`, replace that with `_search.isLoading` and `_search.searchResults`.

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

➤ Change **LandscapeViewController.h** to:

```
@class Search;

@interface LandscapeViewController : UIViewController

@property (nonatomic, strong) Search *search;

@end
```

➤ In **LandscapeViewController.m**, add an import for the `Search` class:

```
#import "Search.h"
```

➤ In `tileButtons`, in the two places where it says `self.searchResults`, replace that with `self.search.searchResults`.

You also need to make a few changes in **Search.m**. You already pasted some of `SearchViewController`'s methods in here but they need a few minor tweaks.

➤ Add an import at the top of **Search.m**:

```
#import "SearchResult.h"
```

➤ In `parseDictionary:`, change the line that adds the object to the `_searchResults` array into:

```
[self.searchResults addObject:searchResult];
```

That is necessary because in this class, `searchResults` is a property instead of a simple instance variable.

OK, that's the first round of changes. Build the app to make sure there are no compiler errors. The app itself doesn't do much anymore because you removed all the searching logic. So let's put that back in.

➤ Add another import to **Search.m**:

```
#import <AFNetworking/AFNetworking.h>
```

➤ Replace `performSearchForText:category:` with:

```
- (void)performSearchForText:(NSString *)text
category:(NSInteger)category
{
  if ([text length] > 0) {
    [queue cancelAllOperations];

    self.isLoading = YES;
    self.searchResults = [NSMutableArray arrayWithCapacity:10];

    NSURL *url = [self urlWithSearchText:text
                                category:category];
    NSURLRequest *request = [NSURLRequest requestWithURL:url];

    AFHTTPRequestOperation *operation = [[AFHTTPRequestOperation
                                alloc] initWithRequest:request];
    operation.responseSerializer =
                           [AFJSONResponseSerializer serializer];

    [operation setCompletionBlockWithSuccess:
       ^(AFHTTPRequestOperation *operation, id responseObject) {

      [self parseDictionary:responseObject];
      [self.searchResults
                     sortUsingSelector:@selector(compareName:)];

      self.isLoading = NO;
      NSLog(@"Success!");

    } failure:^(AFHTTPRequestOperation *operation,
                NSError *error) {

      if (!operation.isCancelled) {
        self.isLoading = NO;
        NSLog(@"Failure!");
      }
    }];

    [queue addOperation:operation];
  }
}
```

This is basically the same thing you did before, except all the user interface logic has been removed. This also uses a `queue` variable that you haven't defined yet. There are several ways you can create this queue (in fact, you can use `AFHTTPRequestOperation` even without making a queue), but I want to show you how to make variables that last even after the object has been deleted. These variables are sometimes called **class variables** because they work at the level of the class itself instead of on just a single instance of that class (this is similar to class methods versus instance methods).

➤ Add the following line below the imports but above the `@interface` line:

```
static NSOperationQueue *queue = nil;
```

This defines a so-called **global** variable. It sits outside any class or method. It works just like any other variable, except that you can use it from anywhere and it keeps its value until the application ends. However, this is not a regular global variable. The keyword `static` restricts its use to just this one source file. Thanks to `static`, you cannot use this `queue` variable anywhere outside **Search.m**.

> **Note:** The keyword `static` has a few different meanings in Objective-C, depending on the context where it is used. Here it means: the visibility of this declaration is limited to just this source file. You've also seen static variables inside of methods before, but that's a different kind of static. Exactly what the `static` keyword does depends on where it is being used, inside or outside of a method.

Where should you initialize this `queue` variable? You only want to do it just once, the very first time it is used. That sounds like a candidate for lazy loading, which will work, but I want to show you another method here.

➤ Add the following method to the `Search` implementation:

```
+ (void)initialize
{
  if (self == [Search class]) {
    queue = [[NSOperationQueue alloc] init];
  }
}
```

This is a class method (notice the `+` in front of its name) and it is invoked the very first time the `Search` class is used in the code. You can think of `initialize` as the init method for the entire class, rather than for a specific instance of that class. If this makes your head spin, then just remember that `initialize` is performed before anything else.

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

That's because the `Search` object 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, blocks are much more convenient. So let's create some blocks.

➤ Add the following line to **Search.h**, above the `@interface` line:

```
typedef void (^SearchBlock)(BOOL success);
```

You've seen the `typedef` statement a few times now. You used it to create enums, for example. It allows you to create a symbolic name for a datatype, in order to save you some typing and to make the code more readable. Here you're declaring a type for your own block, named `SearchBlock`. This is a block that returns no value (`void`) and takes one parameter, a `BOOL` named `success`. 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 `SearchBlock` to refer to a block that takes one `BOOL` parameter and returns no value.

➤ Still in **Search.h**, replace the method signature of `performSearchForText` with:

```
- (void)performSearchForText:(NSString *)text
                    category:(NSInteger)category
                  completion:(SearchBlock)block;
```

You have added a third parameter named `block` that is of type `SearchBlock`. Whoever calls this method can now supply their own block, and the method will execute the code that is inside that block when the search completes. This is done so that the `SearchViewController` can reload its table view and, in the case of an error, show an alert view.

➤ Here are the changes to the method in **Search.m**:

```
- (void)performSearchForText:(NSString *)text
                    category:(NSInteger)category
                  completion:(SearchBlock)block
{
  if ([text length] > 0) {
    . . .

    [operation setCompletionBlockWithSuccess:
       ^(AFHTTPRequestOperation *operation, id responseObject) {

      . . .
```

```
        self.isLoading = NO;
        block(YES);

      } failure:^(AFHTTPRequestOperation *operation,
                  NSError *error) {

        if (!operation.isCancelled) {
          self.isLoading = NO;
          block(NO);
        }
      }];

      . . .
    }
  }
```

It's exactly the same as before but now calls `block(YES)` upon success and `block(NO)` upon failure.

➤ In **SearchViewController.m**, replace `performSearch` with:

```
- (void)performSearch
{
  _search = [[Search alloc] init];
  NSLog(@"allocated %@", _search);

  [_search performSearchForText:self.searchBar.text
    category:self.segmentedControl.selectedSegmentIndex
    completion:^(BOOL success) {
     if (!success) {
       [self showNetworkError];
     }

     [self.tableView reloadData];
  }];

  [self.tableView reloadData];
  [self.searchBar resignFirstResponder];
}
```

You now pass a completion block to the `performSearchForText` method. The code in this block gets called after the search completes, with the `success` parameter being either `YES` or `NO`. A lot simpler than making a delegate, no? This block is always called on the main thread, so it's safe to can use UI code here.

➤ Run the app. You should be able to search again. Notice that the Xcode Debug pane now says:

```
StoreSearch[41868:f803] allocated <Search: 0x6d7a670>
```

When you do a second search, it says:

```
StoreSearch[41868:f803] dealloc <Search: 0x6d7a670>
StoreSearch[41868:f803] allocated <Search: 0x6d84290>
```

The previous `Search` object stays in memory until you perform the next search and then it gets properly deallocated. Nice.

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

Spin me right round

If you flip to landscape while the search is still taking place, the app really ought to show an animated spinner to let the user know something is happening. To do that, you have to check in `viewWillLayoutSubviews` what the state of the active `Search` object is.

➤ In **LandscapeViewController.m**, change `viewWillLayoutSubviews` to:

```
- (void)viewWillLayoutSubviews
{
  [super viewWillLayoutSubviews];

  if (_firstTime) {
    _firstTime = NO;

    if (self.search != nil) {
      if (self.search.isLoading) {
        [self showSpinner];
      } else {
        [self tileButtons];
      }
    }
  }
}
```

If there is no `Search` object then no search has been performed yet and you don't have to do anything. However, if there is a `Search` object and its `isLoading` flag is set, then you need to show the activity spinner.

➤ Add the new `showSpinner` method:

```
- (void)showSpinner
{
  UIActivityIndicatorView *spinner =
       [[UIActivityIndicatorView alloc]
```

```
                initWithActivityIndicatorStyle:
                        UIActivityIndicatorViewStyleWhiteLarge];

  spinner.center = CGPointMake(
                  CGRectGetMidX(self.scrollView.bounds) + 0.5f,
                  CGRectGetMidY(self.scrollView.bounds) + 0.5f);

  spinner.tag = 1000;
  [self.view addSubview:spinner];
  [spinner startAnimating];
}
```

This programmatically creates a new `UIActivityIndicatorView` object (a big white one this time), puts it in the center of the screen and starts animating it. You give the spinner the tag 1000, so you can easily remove it from the screen when the search is done.

➤ Run the app and after starting a search, quickly flip the phone to landscape. You should now see a spinner:

A spinner indicates a search is still taking place

Note: You added `0.5f` to the spinner's `center` position. That is because 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 (240,160) then it would extend 18.5 points to either end. The top-left corner of that spinner is at coordinates (221.5, 141.5), making it look all blurry.

In general you should avoid placing objects at fractional coordinates. By adding 0.5 to both the X and Y, the spinner gets placed at (222, 142) and everything looks sharp. You have to pay attention to this when working with the `center` property and objects that have odd widths or heights.

This is all great, but the spinner doesn't disappear yet when the actual search results are received. The app never notifies the `LandscapeViewController` of this.

There is a variety of ways you can choose to tell the `LandscapeViewController` that the search results have come in, but let's keep it simple and give the view controller a new method.

➤ In **LandscapeViewController.h**, add a new method signature:

```
- (void)searchResultsReceived;
```

➤ Implement this method as follows:

```
- (void)searchResultsReceived
{
  [self hideSpinner];
  [self tileButtons];
}
```

➤ Also add the `hideSpinner` method:

```
- (void)hideSpinner
{
  [[self.view viewWithTag:1000] removeFromSuperview];
}
```

This looks for the view with tag 1000 and then tells that view – the activity spinner – to remove itself from the screen. You could have kept a pointer to the spinner in an instance variable but for a simple situation such as this you might as well use a tag. Because no one else has any strong references to the `UIActivityIndicatorView`, this instance will be deallocated.

The `searchResultsReceived` method should be called from somewhere, of course, and that somewhere is the `SearchViewController`. In the `performSearch` method you provided a completion block to the `Search` object, and that's the ideal place to add this.

➤ Change **SearchViewController.m**'s `performSearch` method to:

```
- (void)performSearch
{
  _search = [[Search alloc] init];

  [_search performSearchForText:self.searchBar.text
    category:self.segmentedControl.selectedSegmentIndex
    completion:^(BOOL success) {
     if (!success) {
       [self showNetworkError];
     }

     [_landscapeViewController searchResultsReceived];
```

```
      [self.tableView reloadData];
    }];

    [self.tableView reloadData];
    [self.searchBar resignFirstResponder];
  }
```

Now this is interesting. When the search starts there is no `_landscapeViewController` object yet because you can only start a search from portrait mode. But by the time the completion block is invoked, the device may have rotated and `_landscapeViewController` will contain a valid pointer.

Upon rotation you also gave `_landscapeViewController` a pointer to the active `Search` object. So 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 `_landscapeViewController` is `nil` and the call to `searchResultsReceived` will simply be ignored.

➤ 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. □

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.

➤ Change the if-statement in `viewWillLayoutSubviews` to the following:

```
    if (self.search != nil) {
      if (self.search.isLoading) {
        [self showSpinner];
      } else if ([self.search.searchResults count] == 0) {
        [self showNothingFoundLabel];
      } else {
        [self tileButtons];
      }
    }
```

You've added one condition: if there are no items in the `searchResults` array, you'll call the `showNothingFoundLabel` method.

➤ Here is that method:

```
- (void)showNothingFoundLabel
```

```
{
  UILabel *label = [[UILabel alloc] initWithFrame:CGRectZero];
  label.text = @"Nothing Found";
  label.backgroundColor = [UIColor clearColor];
  label.textColor = [UIColor whiteColor];

  [label sizeToFit];
  CGRect rect = label.frame;
  rect.size.width = ceilf(rect.size.width/2.0f) * 2.0f;
  rect.size.height = ceilf(rect.size.height/2.0f) * 2.0f;
  label.frame = rect;
  label.center = CGPointMake(
            CGRectGetMidX(self.scrollView.bounds),
            CGRectGetMidY(self.scrollView.bounds));

  [self.view addSubview:label];
}
```

Here you first create a `UILabel` object by hand and give it text and color. It's important that the `backgroundColor` property is set to `[UIColor clearColor]` to make it see-through. Then you tell the label to resize itself to the optimal size using `sizeToFit`. You could have given the label a frame that was big enough to begin with, but I find this just as easy.

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 = ceilf(width/2.0f) * 2.0f;
```

If you divide a number such as 11 by 2 you get 5.5 (provided they're not all integers, but both `width` and `2.0f` are `floats` so you're good). The `ceilf()` function rounds 5.5 up to make 6, and then you multiply by 2.0 to get a final value of 12. This formula always gives you the next even number if the original is odd.

> **Note:** Because you're not using a hardcoded number such as 480 or 568 but `self.scrollView.bounds` to determine the width of the screen, the code to center the label works correctly on both 3.5-inch and 4-inch devices.

▸ Run the app and search for something ridiculous (**ewdasuq3sadf843** will do). When the search is done, flip to landscape.

Yup, nothing found here either

It doesn't work properly yet when you flip to landscape while the search is taking place. Of course you also need to put some logic in `searchResultsReceived`.

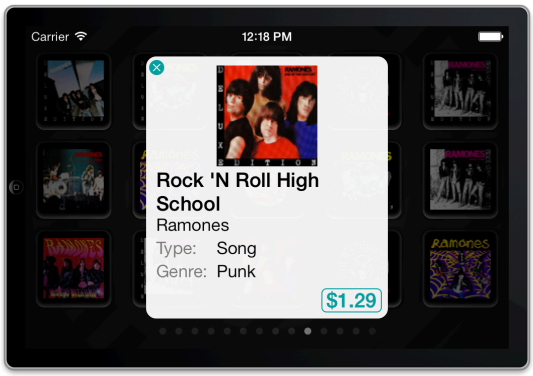
▸ Change that method to:

```
- (void)searchResultsReceived
{
  [self hideSpinner];

  if ([self.search.searchResults count] == 0) {
    [self showNothingFoundLabel];
  } else {
    [self tileButtons];
  }
}
```

Now you should have all your bases covered.

The Detail pop-up

These things are not buttons for nothing. The app should show the Detail pop-up when you tap them, like this:



The pop-up in landscape mode

This is fairly easy to achieve. When adding the buttons you can give them a **target-action**, i.e. a method to call when the Touch Up Inside event is received. Just like in Interface Builder, except now you hook up the event to the action method programmatically.

➤ Add the following two lines to the button creation code in `tileButtons`:

```
button.tag = 2000 + index;
[button addTarget:self action:@selector(buttonPressed:)
             forControlEvents:UIControlEventTouchUpInside];
```

First you give the button a tag, so you know to which index in the `searchResults` array this button corresponds. You need that in order to find the correct `SearchResult` object. You also tell the button it should call the `buttonPressed:` method when it gets tapped.

Tip: I 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.

➤ Add the `buttonPressed:` method to the view controller:

```
- (void)buttonPressed:(UIButton *)sender
{
  DetailViewController *controller = [[DetailViewController
     alloc] initWithNibName:@"DetailViewController" bundle:nil];

  SearchResult *searchResult =
                   self.search.searchResults[sender.tag - 2000];
  controller.searchResult = searchResult;

  [controller presentInParentViewController:self];
}
```

This is very similar to what you did in `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).

Notice that this is an action method but you didn't declare it as `IBAction`. That is only necessary when you want to connect the method to something in Interface Builder. Here you made the connection programmatically, so you can skip `IBAction` and simply make the method `void`.

Exercise. This doesn't compile. Why not? □

Ha ha, just checking if you're paying attention. You still need to import the header for `DetailViewController`.

➤ Run the app and check it out.

In the debug pane output you should see that the `DetailViewController` is properly deallocated when you rotate back to portrait.

› If you're happy with the way it works, then let's commit it. If you also made a branch, then merge it back into the master branch.

You can find the project files for the app under **10 - Refactored Search** in the tutorial's Source Code folder.

Compiler warning flags

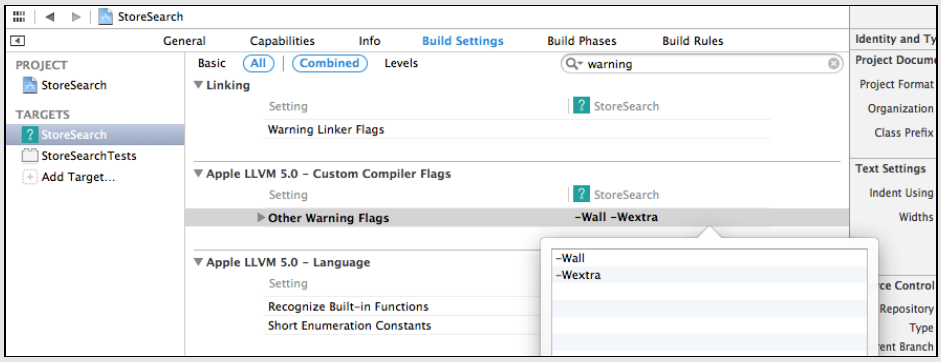
The Build Settings tab in the Project Settings screen contains a ton of configuration options that influence how your app will be compiled. Of special importance are the *warning flags*.

› In the **Project Settings** screen, switch to the **Build Settings** tab. Click on **All** (next to "Basic") to show all the options.

› In the search box at the top, enter **warning** to narrow down the list.

This shows all the potential programming mistakes that Xcode can warn you about. It's wise to enable as many of these warnings as possible.

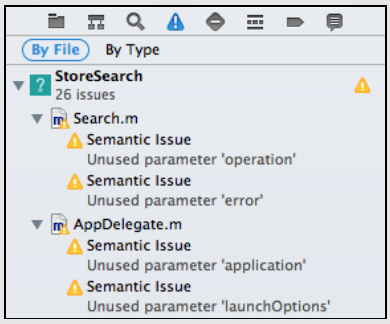
› Change the value of the **Other Warning Flags** setting to **-Wall -Wextra**. This means you want the compiler to warn about anything that it finds suspicious. By default the compiler only checks for certain types of errors but you want it to check for everything as that's the best way to prevent silly programming mistakes.



› Now search for **analyzer**. Change the **Analyze During Build** setting to **Yes**. For extra thoroughness, change **Mode of Analysis for Build** to **Deep**.

The Static Analyzer goes a step beyond what the regular compiler checks for and it can find serious errors in your program. You can run the Static Analyzer by hand at any time by holding down on the Run button and then picking Analyze from the pop-up. By enabling this option in the Build Settings, Xcode will automatically run the analyzer every time you build the app. I recommend you enable the Static Analyzer on all your projects.

➤ Build the app. You should get a whole bunch of compiler warnings that you didn't get before:



These don't look like serious errors. All of them are "Unused parameter" issues, which means some of your methods have parameters that they don't do anything with. That's not uncommon, and fortunately you can make an exception so Xcode stops giving these warnings.

➤ Add **-Wno-unused-parameter** to the Other Warning Flags. Build the app again and there should be no more issues.

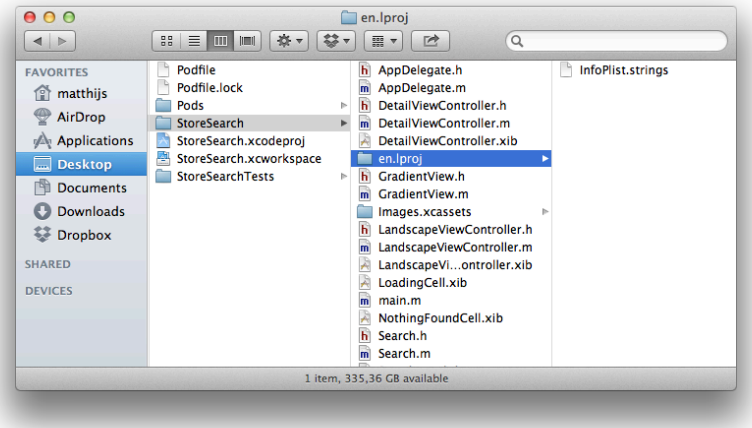
Internationalization

So far the apps you've made in this tutorial series have all been in English. No doubt the United States are the single biggest market for apps, but if you add up all the countries where English isn't the primary language then you end up with quite a sizable market that you might be missing out on.

Fortunately, iOS makes it very easy to add support for other languages to your apps, a process known as **internationalization**. This is often abbreviated as "i18n" because that's a lot shorter to write (the 18 stands for the number of letters between the i and the n). You'll also often hear the word **localization**, which basically means the same thing.

In this section you'll add support for Dutch, which is my native language. You'll also make the web service query return results that are optimized for the user's regional settings.

The structure of your source code folder probably looks something like this:

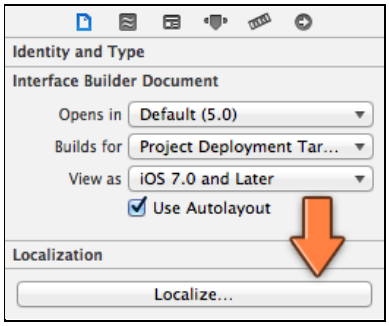


The files in the source code folder

There is a subfolder named **en.lproj** that contains one file, **InfoPlist.strings**. The en.lproj folder contains files that are localized for the English language. 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 (**nl** in the case of Dutch).

Let's begin by localizing a simple file, the **NothingFoundCell.xib**. Often nib files will contain text that needs to be translated. You can simply make a new copy of the existing nib file for a specific language and put it in the right .lproj folder. When the iPhone is using that language, it will automatically load the translated nib.

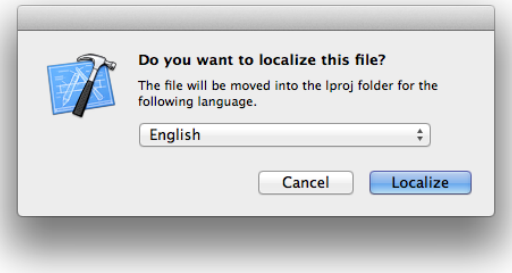
› Select **NothingFoundCell.xib** in the Project navigator. Switch to the **File inspector** pane (on the right of the Xcode window). Because the NothingFoundCell.xib file isn't in any XX.lproj folders, it does not have any localizations yet.



The NothingFoundCell has no localizations

› Click the **Localize...** button in the Localization section.

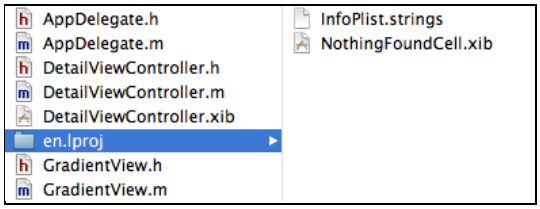
Xcode asks for confirmation because this involves moving the file to a new folder:



Xcode asks whether it's OK to move the file

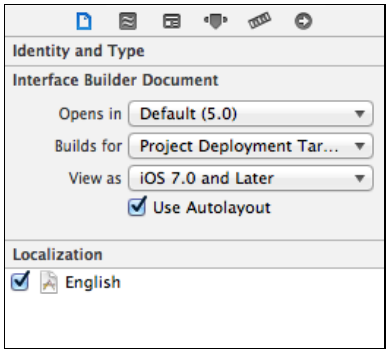
› Click **Localize** to continue.

Look in Finder and you will see that NothingFoundCell.xib has moved to the **en.lproj** folder:



Xcode moved NothingFoundCell.xib to the en.lproj folder

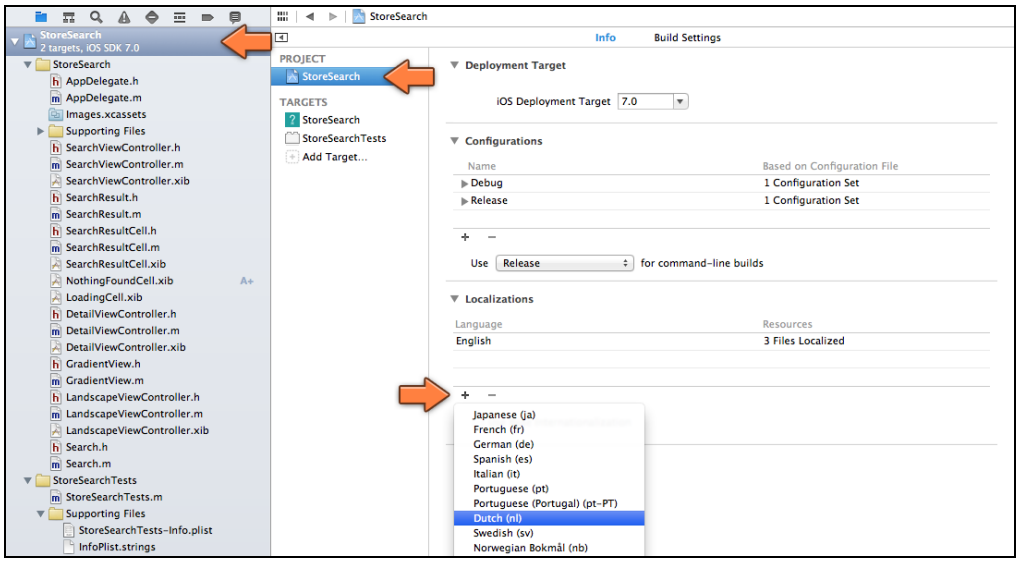
If you select **NothingFoundCell.xib** again in the Project navigator, you will see that the **File inspector** now lists English as one of the localizations.



The Localization section now contains an entry for English

To add a new language you have to switch to the **Project Settings** screen.

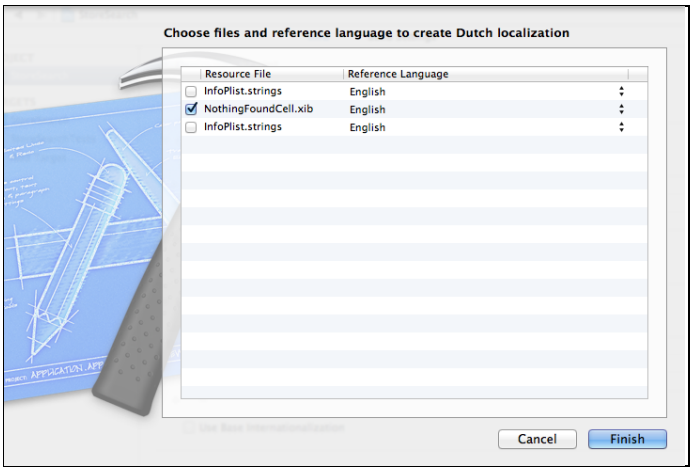
› In the **Project Settings** screen, select the project (not one of the targets), and press the **+** button in the Localizations section:



Adding a new language

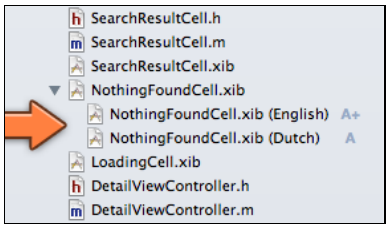
› From the pop-up menu choose **Dutch (nl)**.

Xcode now asks which resources you want to localize. Uncheck everything except for **NothingFoundCell.xib** and click **Finish**.



Choosing the files to localize

If you look in Finder again you'll notice that a new subfolder has been added, **nl.lproj**, and that it contains another copy of NothingFoundCell.xib. That means there are now two nib files for `NothingFoundCell`. You can also see this in the Project navigator:



NothingFoundCell.xib has two localizations

Let's edit the new Dutch version of this nib.

› Click on **NothingFoundCell.xib (Dutch)** to open it in Interface Builder. Change the label text to **Niets gevonden** and center the label again in the view.

That's how you say it in Dutch

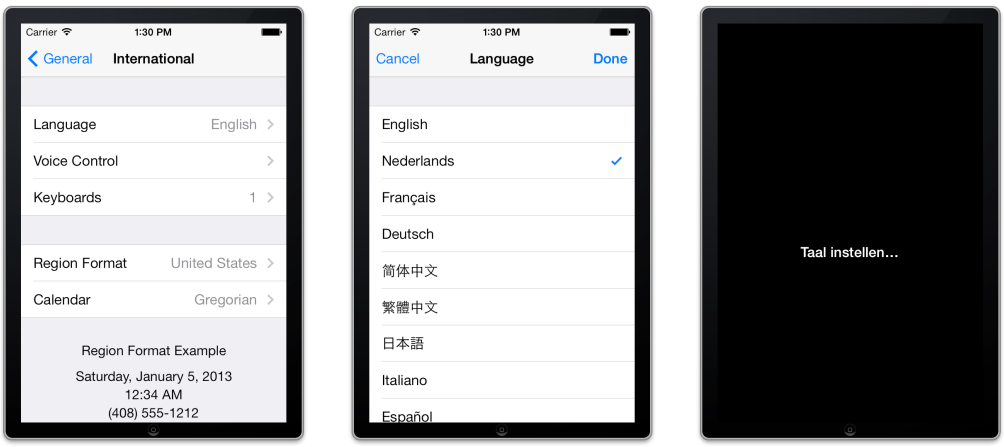
It is perfectly all right to resize or move around items in a translated nib. You could make the whole nib look completely different if you wanted to (but that's probably a bad idea). Some languages, such as German, have very long words and in those cases you may have to tweak label sizes and fonts to get everything to fit.

If you run the app now, nothing will have changed. You have to switch the Simulator to use the Dutch language first. However, before you do that you really should remove the app from the simulator, clean the project, and do a fresh build. The reason for this is that the nibs were previously not localized. If you were to switch the simulator's language now, the app would still keep using the old, non-localized versions of the nibs.

> **Note:** For this reason it's a good idea to already put all your nib files and storyboards in the **en.lproj** folder when you create them (or in Base.lproj, which we'll discuss shortly), even if you don't intend to internationalize your app any time soon, just so users won't run into the same problem. You don't want to ask your users to uninstall the app – and lose their data – in order to be able to switch languages.

› Remove the app from the Simulator. Do a clean (**Product** → **Clean** or **Shift-⌘-K**) and re-build the app.

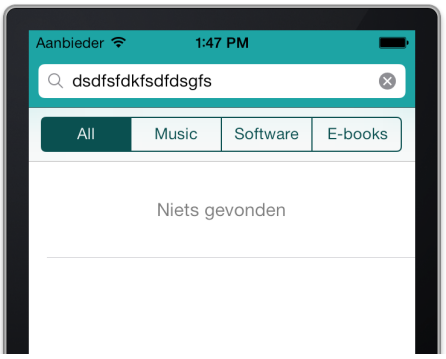
› Open the **Settings** app in the Simulator and go to **General** → **International** → **Language**. From the list pick **Nederlands**.



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 **.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 **DerivedData** folder. Empty your trashcan. Then open the StoreSearch workspace again and give it another try. (Don't forget to switch the Simulator back to **Nederlands**.)

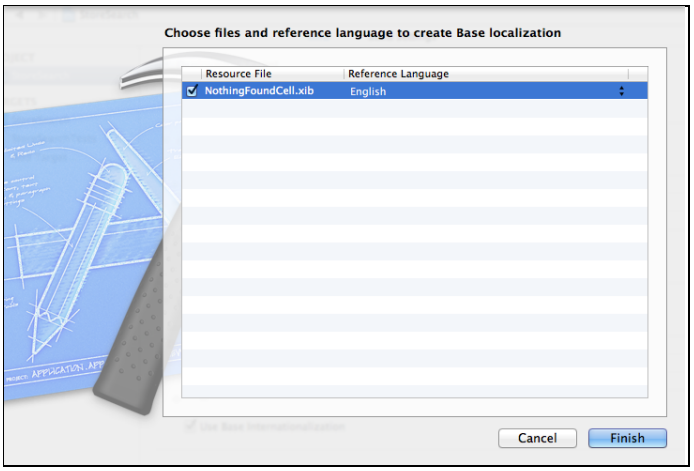
To localize the other nibs you could repeat the process and add copies of their xib files to the nl.lproj folder. That isn't too bad for this app but if you have an app with a really complicated screen layout then having multiple copies of the same nib can become a maintenance nightmare. Whenever you need to change something to

that screen you also need to update all of the nibs. There's a risk that you forget one nib and they go out-of-sync. That's just asking for bugs – in languages that you probably don't speak!

To prevent this from happening you can use **base internationalization**. With this feature enabled you don't copy the entire nib, but only the texts.

› Go to the **Project Settings** screen and check the **Use Base Internationalization** option:

Xcode pops up a dialog to ask which language it should base this on. The project only has one localized nib, so that's an easy choice:

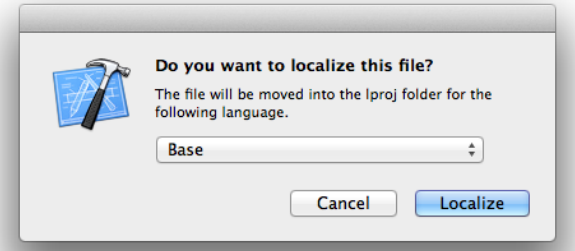


Choosing the reference language for the base localization

› Look at the project folder in Finder. It now has a new subfolder, **Base.lproj**, and the **NothingFoundCell.xib** got moved from **en.lproj** into this new folder. However, **nl.proj** still has its own copy. That's fine for now.

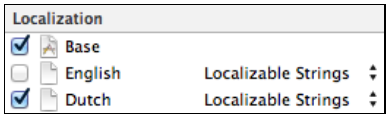
Let's translate the other nibs.

› Open **LoadingCell.xib** in Interface Builder. In the **File inspector** press the **Localize...** button. Xcode again asks for confirmation; this time use **Base** as the language:



Choosing the Base localization as the destination

› Select **LoadingCell.xib** again in the Project navigator. The Localization section in the **File inspector** now contains three options: Base (with a checkmark), English, and Dutch. Put a checkmark in front of **Dutch**:

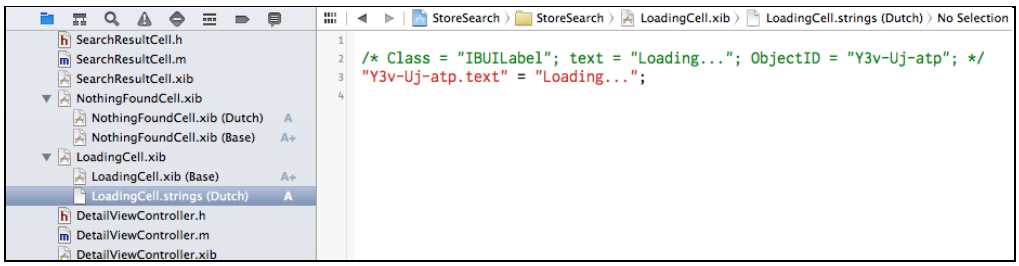


Adding a Dutch localization

In Finder you can see that **nl.proj** now doesn't get a copy of the nib, but a new type of file: **LoadingCell.strings**.

› Expand **LoadingCell.xib** 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

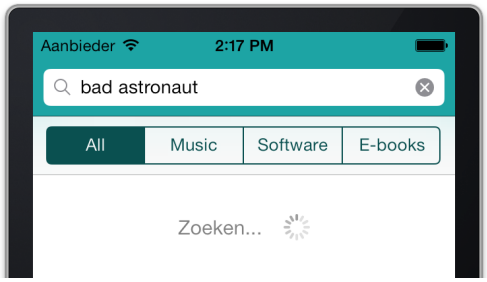
There is still only one nib, the one from the Base localization. But the Dutch translation consists of a "strings" file with just the texts from the labels, buttons, and other controls.

The contents of this particular strings file are:

```
/* Class = "IBUILabel"; text = "Loading..."; ObjectID = "Y3v-Uj-atp"; */
"Y3v-Uj-atp.text" = "Loading...";
```

The green bit is a comment, just like in Objective-C. The second line says that the **text** property of the object with ID "Y3v-Uj-atp" contains the text **Loading...** That 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.

› Change the text **Loading...** into **Zoeken...** 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 foler, reset the Simulator. Hopefully by the time you read this tutorial, this very annoying bug has been exterminated from Xcode.

› Repeat the steps to make a new Base localization for **SearchViewController.xib**, and then add a Dutch localization as well.

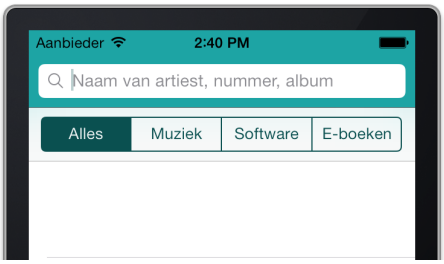
For the `SearchViewController` screen two things need to change: the placeholder text in the Search Bar and the labels on the Segmented Control.

› In **SearchViewController.strings (Dutch)** change the placeholder text to **Naam van artiest, nummer, album**.

The segment labels will become: **Alles**, **Muziek**, **Software**, and **E-Boeken**.

```
"YD3-Lf-PIq.segmentTitles[0]" = "Alles";

"YD3-Lf-PIq.segmentTitles[1]" = "Muziek";

"YD3-Lf-PIq.segmentTitles[2]" = "Software";

"YD3-Lf-PIq.segmentTitles[3]" = "E-boeken";

"i96-Ql-AEj.placeholder" = "Naam van artiest, nummer, album";
```

(Of course your object IDs will be different.)



The localized SearchViewController

> **Note:** The search bar's placeholder text did not get translated for me when I ran the app with the Simulator set to Dutch (I faked it in the screenshot). This looks like a bug in iOS 7.
>
> Just so you know, there is a workaround: replace the strings file with a translated xib. You can change this in the File inspector's Localization section. Next to each language is a control that says Localized Strings; change this into Interface Builder Cocoa Touch XIB, and this will place a localized xib file in the language's .lproj folder. That's also how the NothingFoundCell.xib is currently set up.

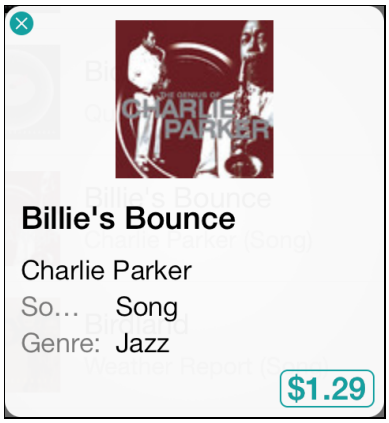
- Create Base and Dutch localizations for **DetailViewController.xib**.
- Change the strings file to:

```
// Used to be "Type:"
"HEX-Ur-db7.text" = "Soort:";

// Don't need to change these:
"AC5-Yr-ahB.text" = "Genre:";
"6wl-9Y-9fy.text" = "Kind Value";
"JgW-rH-rTG.text" = "Artist Name";
"b4u-9x-hY7.text" = "Name";
"eBR-nx-VHc.text" = "Genre Value";
"DJG-Bx-u1s.normalTitle" = "$9.99";
```

You only need to change the **Type:** label to say **Soort:**. All the other labels can remain the same because you will replace them with values from the `SearchResult` object anyway. ("Genre" is the same in both languages.)

You can even remove the texts that don't need localization. If a localized version for a specific resource is missing for the user's language, then iOS will fall back to the one from the Base localization.

When you run the app there's a small problem. The text "Soort:" doesn't fit into the label:

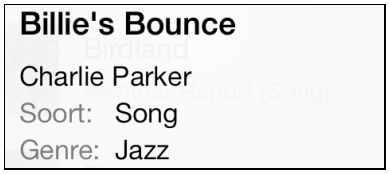


The "Soort:" label is cut off

That's a common issue with localization. English words tend to be shorter than words in many other languages so you have to make sure your labels are big enough.

› Open **DetailViewController.xib (Base)** and make the **Type:** label a bit wider.

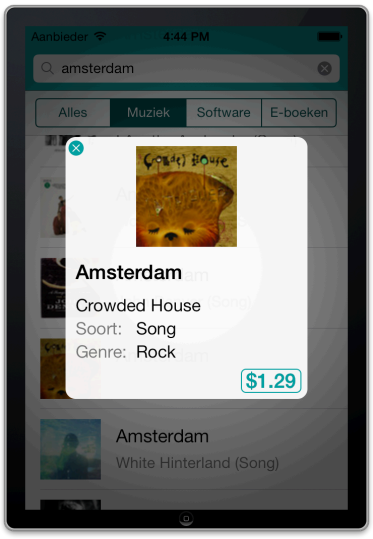
That's better:



Now the label is big enough

› There is no real need to give **LandscapeViewController.xib** and **SearchResult.xib** a Dutch localization (there is no on-screen text in the nib itself) but do give them a Base localization. This prepares the app for the future, should you need to localize these nibs at some point.

When you're done there should be no more **xib** files outside the **.lproj** folders.



The app in Dutch

That's it for the nibs. Not so bad, was it? I'd say all these changes are commit-worthy.

Localizing on-screen texts

Even though the nibs have been translated, not all of the text is. For example, in the image above the text from the `kind` property is still "Song". While in this case you could get away with it – probably everyone in the world knows what the word "Song" means –not all of the texts from the `kindForDisplay` method will be understood by non-English speaking users.

To localize texts that are not in a nib, you have to use another approach.

› In **SearchResult.m**, replace the `kindForDisplay` method with:

```
- (NSString *)kindForDisplay
{
  if ([self.kind isEqualToString:@"album"]) {
    return NSLocalizedString(@"Album",
                             @"Localized kind: Album");
  } else if ([self.kind isEqualToString:@"audiobook"]) {
    return NSLocalizedString(@"Audio Book",
                             @"Localized kind: Audio Book");
  } else if ([self.kind isEqualToString:@"book"]) {
    return NSLocalizedString(@"Book", @"Localized kind: Book");
  } else if ([self.kind isEqualToString:@"ebook"]) {
    return NSLocalizedString(@"E-Book",
                             @"Localized kind: E-Book");
  } else if ([self.kind isEqualToString:@"feature-movie"]) {
    return NSLocalizedString(@"Movie",
```

```
                                @"Localized kind: Feature Movie");
  } else if ([self.kind isEqualToString:@"music-video"]) {
    return NSLocalizedString(@"Music Video",
                             @"Localized kind: Music Video");
  } else if ([self.kind isEqualToString:@"podcast"]) {
    return NSLocalizedString(@"Podcast",
                                @"Localized kind: Podcast");
  } else if ([self.kind isEqualToString:@"software"]) {
    return NSLocalizedString(@"App",
                                @"Localized kind: Software");
  } else if ([self.kind isEqualToString:@"song"]) {
    return NSLocalizedString(@"Song", @"Localized kind: Song");
  } else if ([self.kind isEqualToString:@"tv-episode"]) {
    return NSLocalizedString(@"TV Episode",
                             @"Localized kind: TV Episode");
  } else {
    return self.kind;
  }
}
```

The structure is still the same as before, but instead of doing,

```
return @"Album";
```

it now does:

```
return NSLocalizedString(@"Album", @"Localized kind: Album");
```

This is slightly more complicated but it is also a lot more powerful. `NSLocalizedString()` takes two parameters: the text to return (`@"Album"`) and a comment (`@"Localized kind: Album"`).

Here is the cool thing: if your app includes a file named **Localizable.strings** for the user's language, then `NSLocalizedString()` will look up the text (`@"Album"`) and returns the translation as specified in Localizable.strings.

If no translation for that text is present, or there is no Localizable.strings file, then `NSLocalizedString()` simply returns the text as-is.

➤ Run the app again. The "Type:" field in the pop-up (or "Soort:" in Dutch) should still show the same kind of texts as before because you haven't translated anything yet.

To create the **Localizable.strings** file, you will use a command line tool named **genstrings**. This requires a trip to the Terminal.

➤ Open a Terminal, `cd` to the folder that contains the StoreSearch project. You want to go into the folder that contains the actual source files. On my system that is:

```
cd ~/Desktop/StoreSearch/StoreSearch
```

Then type the following command:

```
genstrings *.m -o en.lproj
```

This looks at all your source files (***.m**) and writes a new file called **Localizable.strings** in the **en.lproj** folder.

➤ Add this **Localizable.strings** file to the project in Xcode. I like to put it under the **Supporting Files** group.

If you open the Localizable.strings file, this is what it currently contains:

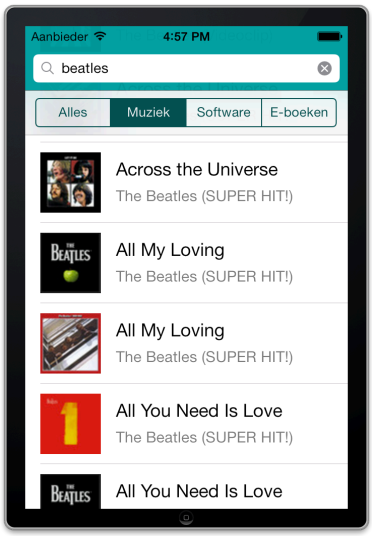
```
/* Localized kind: Album */
"Album" = "Album";

/* 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.

➤ Change the "Song" line to:

```
"Song" = "SUPER HIT!";
```

➤ Now run the app again and search for music. For any search result that is a song, it will now say "SUPER HIT!" instead.



Where it used to say Song it now says SUPER HIT!

Of course, changing the texts in the English localization doesn't make much sense, so put Song back to what it was and then we'll do it properly.

➤ In the **File inspector**, add a Dutch localization for this file.

➤ 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 whole 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 had to do was run the genstrings tool and you'd get all the strings. Now you have to comb through the source code and add `NSLocalizedString()` everywhere there is text that will be shown to the user.

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 make more money, 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
artistName = @"Unknown";

priceText = @"Free";


// SearchResultCell, configureForSearchResult
artistName = @"Unknown";


// LandscapeViewController, showNothingFoundLabel
label.text = @"Nothing Found";


// SearchViewController, showNetworkError
initWithTitle:@"Whoops..."

message:@"There was an error reading from the iTunes Store.
          Please try again."

cancelButtonTitle:@"OK"
```

➤ Add `NSLocalizedString()` around these texts. For example, in the `UIAlertView` constructor in `showNetworkError`, you could write:

```
initWithTitle:NSLocalizedString(
                          @"Whoops...", @"Error alert: title")
```

Note: You don't need to use `NSLocalizedString()` for debug output with your `NSLog()`s. Debug output is really intended only for you, the developer, so it's best if it is in English.

➤ Run the **genstrings** tool again. Give it the same arguments as before. It will put a clean file with all the new strings in the **en.lproj** folder.

Unfortunately, there really isn't a good way to make genstrings merge new strings into existing translations. It will overwrite your entire file and throw away any changes that you made. There is a way to make the tool append its output to an existing file but then you end up with a lot of duplicate strings.

Tip: Always regenerate only the file in en.lproj and then copy over the missing strings to your other Localizable.strings files. You can use a tool such as FileMerge or Kaleidoscope to compare the two to see where the new strings are. There are also several third-party tools on the Mac App Store that are a bit friendlier to use than genstrings. I am partial to Linguan (http://www.cocoanetics.com/apps/linguan/).

➤ Add these new translations to the Dutch **Localizable.strings**:

```
"Nothing Found" = "Niets gevonden";

"There was an error reading from the iTunes Store. Please try again." =
"Er ging iets fout bij het communiceren met de iTunes winkel. Probeer
het nog eens.";

"Unknown" = "Onbekend";

"Whoops..." = "Foutje...";
```

It may seem a little odd that such as long string as "There was an error reading from the iTunes Store. Please try again." would be used as the lookup key for this file, but there really isn't anything wrong with it.

Some people write code like this:

```
NSString *s = NSLocalizedString(
            @"ERROR_MESSAGE23", @"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 and it requires that you always have an English Localizable.strings as well. In any case, you will see both styles used in practice.

Note also that the text `@"Unknown"` occurred only once in Localizable.strings even though it shows up in two different places in the source code. Each piece of text only needs to be translated once.

If your app builds strings dynamically, then you can also localize these texts. For example in `SearchResultCell`, `configureForSearchResult:` you do:

```
self.artistNameLabel.text =
       [NSString stringWithFormat:@"%@ (%@)", artistName, kind];
```

You could internationalize this as follows:

```
self.artistNameLabel.text = [NSString stringWithFormat:
      NSLocalizedString(@"%@ (%@)",
            @"Format for artist name label"), artistName, kind];
```

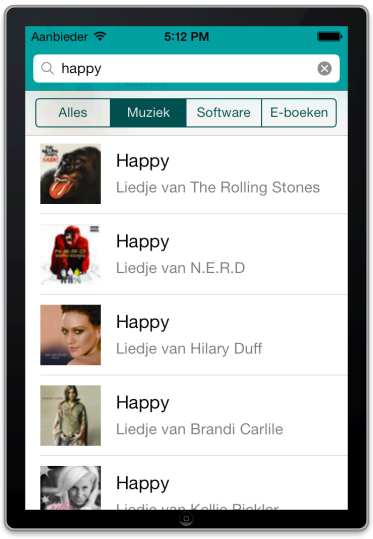
This shows up in Localizable.strings as:

```
/* Format for artist name label */
"%@ (%@)" = "%1$@ (%2$@)";
```

If you wanted to, you could change the order of these parameters in the translated file. For example:

```
"%@ (%@)" = "%2$@ van %1$@";
```

It will turn the artist name label into something like this:



There are a lot of songs named Happy

In this circumstance I would advocate the use of a special key rather than the literal string to find the translation. It's thinkable that your app will employ the format string `"%@ (%@)"` in some other place and you may want to translate that completely differently there. I'd call it something like `"ARTIST_NAME_LABEL_FORMAT"` instead:

```
/* 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 that in the code as well:

```
self.artistNameLabel.text = [NSString stringWithFormat:
     NSLocalizedString(@"ARTIST_NAME_LABEL_FORMAT",
            @"Format for artist name label"), artistName, kind];
```

➤ Do I need to say it? Commit!

InfoPlist.strings

Some apps also have a different name depending on the user's language. If you've ever wondered what the **InfoPlist.strings** file is for, then this is it. It contains localized versions of the keys from your Info.plist file.

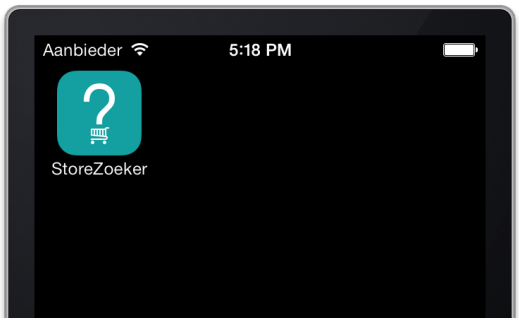
➤ Add a Dutch localization for **InfoPlist.strings** (this file is listed under the Supporting Files group).

➤ Open the Dutch version and add the following line:

```
/* Localized versions of Info.plist keys */
CFBundleDisplayName = "StoreZoeker";
```

(If you're Dutch, sorry for the stupid name. This is the best I could come up with.)

➤ 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. That has two reasons:

1. The language settings are independent of the regional settings. How currencies and numbers are displayed depends on the region settings, not the language.
2. The app does not specify anything about country or language when it sends the requests to the iTunes store, so the web service always returns prices in US Dollars.

First you'll fix the app so that it sends information about the user's language and regional settings to the iTunes store. The method that you are going to change is **Search.m**'s `urlWithSearchText:` because that's where you construct the parameters that get sent to the web service.

➤ Change the `urlWithSearchText:` method to the following:

```
- (NSURL *)urlWithSearchText:(NSString *)searchText
                    category:(NSInteger)category
{
  NSString *categoryName;
  switch (category) {
    case 0: categoryName = @""; break;
```

```
    case 1: categoryName = @"musicTrack"; break;
    case 2: categoryName = @"software"; break;
    case 3: categoryName = @"ebook"; break;
  }

  NSLocale *locale = [NSLocale autoupdatingCurrentLocale];
  NSString *language = [locale localeIdentifier];
  NSString *countryCode = [locale objectForKey:
                                        NSLocaleCountryCode];

  NSString *escapedSearchText = [searchText
stringByAddingPercentEscapesUsingEncoding:NSUTF8StringEncoding];

  NSString *urlString = [NSString stringWithFormat:
@"http://itunes.apple.com/search?term=%@&limit=200&entity=%@&lang=%@&cou
ntry=%@",
        escapedSearchText, categoryName, language, countryCode];

  NSURL *url = [NSURL URLWithString:urlString];

NSLog(@"URL: %@", url);
  return url;
}
```

The regional settings are also referred to as the user's **locale** and of course there is an object for it, `NSLocale`. You get a reference to the `autoupdatingCurrentLocale`. 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 his regional information while the app is running, the app will automatically use these new settings the next time it does something with that `NSLocale` 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. I also added an `NSLog()` for good measure so you can see what exactly the URL will be.

➤ Run the app and do a search. Xcode should output the following:

```
http://itunes.apple.com/search?term=bird&limit=200&entity=&lang=en_US&co
untry=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.

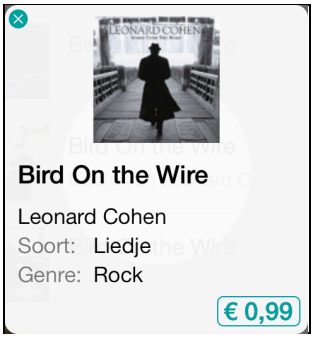
➤ While keeping the app running, switch to the Settings app to change the regional settings. If the Simulator is still in Dutch, then it is under **Algemeen** →

Internationaal → **Regionotatie**. Change it to **Nederland**, then switch back to StoreSearch and repeat the search.

Xcode now says:

```
http://itunes.apple.com/search?term=bird&limit=200&entity=&lang=nl_NL&co
untry=NL
```

The language and country have both been changed to NL (for the Netherlands). If you tap on a search result you'll see that the price is now in Euros:



The price according to the user's region settings

Of course, you have to thank `NSNumberFormatter` 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 will take only a small bit of effort that definitely pays back. (You can put the Simulator back to English now.)

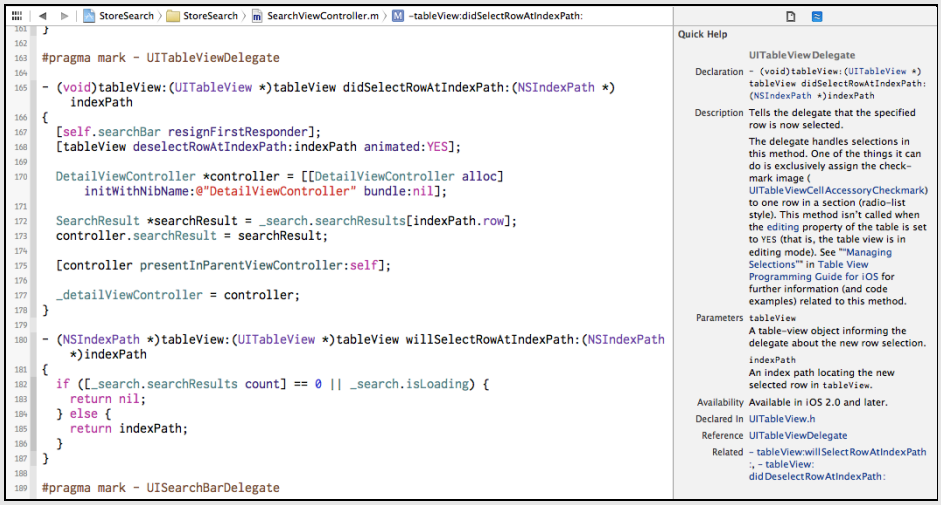
› It's time to commit because you're going to make some big changes in the next section. If you've also been tagging the code, you can call this v0.9, as you're rapidly approaching the 1.0 version that is ready for release.

The project files for the app up to this point are under **11 - Internationalization** in the tutorial's Source Code folder.

The documentation

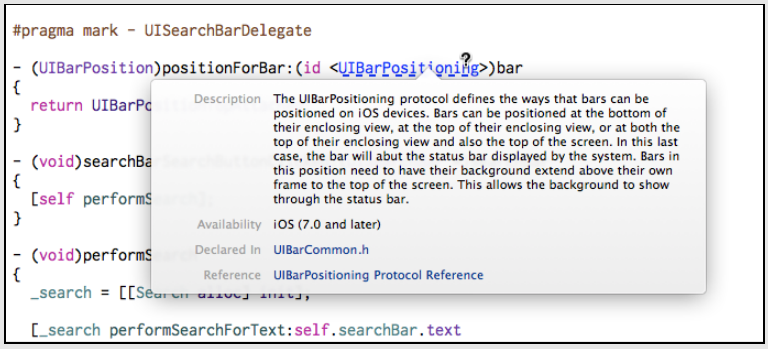
Xcode comes with a big library of documentation for developing iOS apps. Basically everything you need to know is in here. Learn to use the Xcode Documentation browser – it will become your best friend!

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

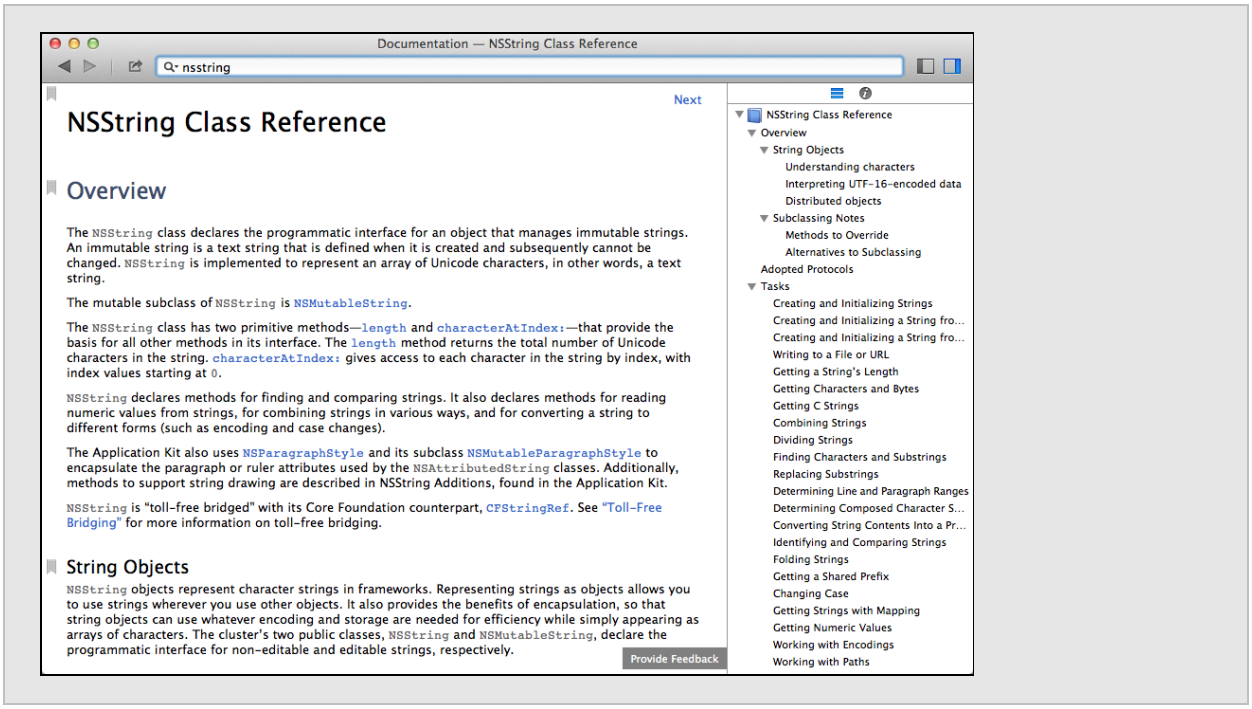


Simply have the Quick Help inspector open (the second tab in the inspector pane) and it will show context-sensitive help. Put the text cursor on the thing you want to know more about and the inspector will give a summary on it. You can click any of the blue text to jump to the full documentation.

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



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

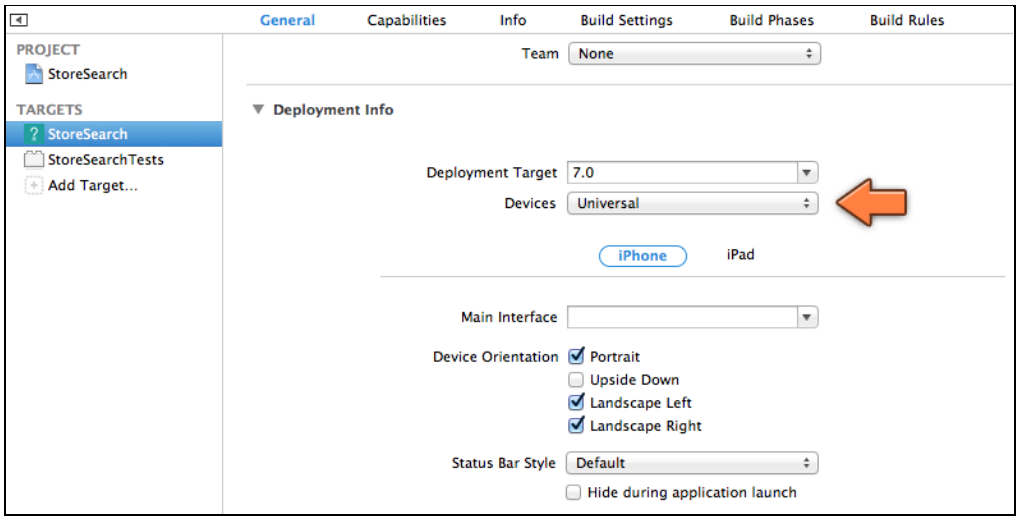


The iPad

Even though the apps you've written so far are only for the iPhone, everything you have learned also applies to writing iPad apps. There really isn't much difference between the two: they both run iOS and have access to the exact same frameworks. But the iPad has a much bigger screen (768×1024 points instead of 320×568) and that makes all the difference.

In this section you'll make the app **universal** so that it runs on both the iPhone and the iPad. You are not required to always make your apps universal; you can also make apps that run only on the iPad and not on the iPhone.

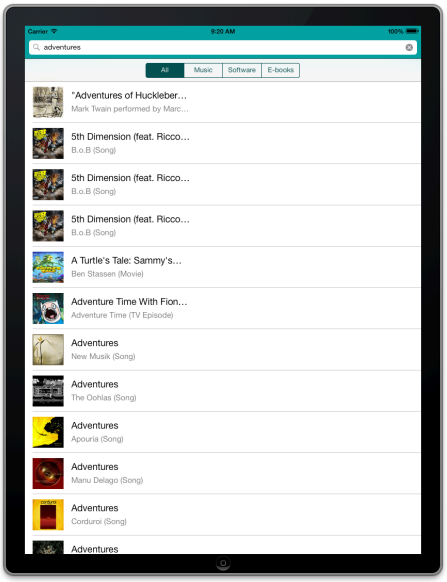
› Go to the **Project Settings** screen and select the StoreSearch target. In the **General** tab, under **Deployment Info** there is a setting for **Devices**. It is currently set to iPhone, but change it to **Universal**.



Making the app universal

That's enough to make the app run on the iPad.

▸ Choose the **iPad** Simulator in the box in the top-left corner of the Xcode window and run the app.



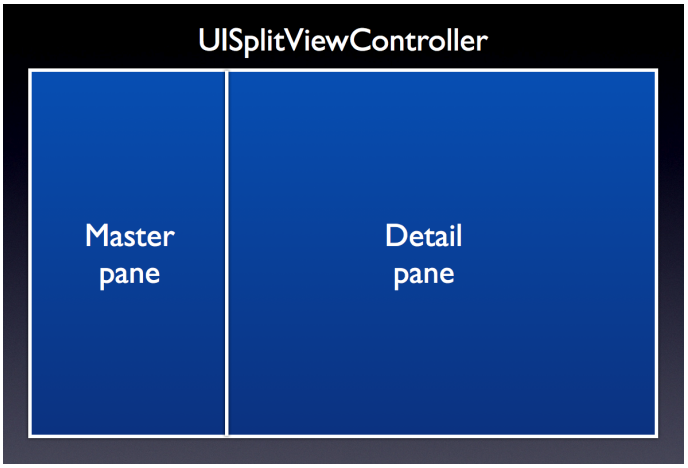
StoreSearch in the iPad Simulator

This works but simply blowing up the interface to iPad size is not taking advantage of all the extra space the bigger screen offers. So instead you're going to use some of the special features that UIKit has to offer on the iPad, such as the split-view controller and popovers.

The split-view controller

On the iPhone, a view controller basically manages the whole screen, although you've seen that you can embed a view controller inside another. On iPad it is common for view controllers to manage just a section of the screen, because the display is so much bigger and often you will want to combine different types of content in the same screen.

A good example of this is the split-view controller. It has two panes, a big one and a smaller one. The smaller pane is on the left (the "master" pane) and usually shows a list. The right pane (the "detail" pane) shows more information about the thing you have selected in the master list. If you've used an iPad before then you've seen the split-view controller in action because it's used in many of the standard apps such as Mail and Settings.



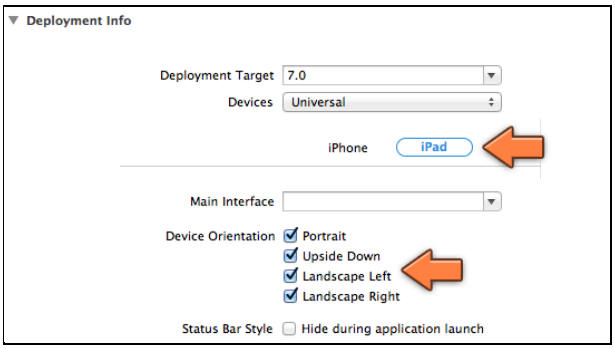
The split-view controller has a master pane and detail pane, both of which contain their own view controllers

In this section you'll convert the app to use such a split-view controller. This has some consequences for the organization of the user interface.

Because the iPad has different dimensions from the iPhone it will also be used in different ways. Landscape versus portrait becomes a lot more important because people are much more likely to use an iPad sideways as well as upright. Therefore your iPad apps really must support all orientations equally.

This implies that you shouldn't make landscape show a completely different UI than portrait, so what you did with the iPhone version of the app won't fly on the iPad – you can no longer show the `LandscapeViewController` when the user rotates the device. That feature goes out of the window.

➤ In the Deployment Info section there is now a tab for iPad. Here you can change some of your app's settings specifically for running on the iPad. Enable all the device orientations:



iPad apps should support all orientations

Let's put that split-view controller into the app. You will be creating it in `AppDelegate` when the app starts up.

➤ Add a new property to **AppDelegate.h**:

```
@property (strong, nonatomic) UISplitViewController
                                        *splitViewController;
```

A split-view controller takes 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. However, neither of these classes was designed with the split view in mind so you need to make a few changes to accommodate this.

If the iPad is in landscape, the split-view controller has enough room to show both panes at the same time. However, in portrait mode only the detail view controller is visible. Your app has to provide a button that will slide the master pane into view. (You can also swipe the screen to reveal and hide it.)

To make this work, the split-view controller requires a delegate (not too surprising, I hope) to implement certain methods. In this app, you'll make `DetailViewController` assume these delegate duties.

The split-view controller in landscape and portrait orientations

› In **AppDelegate.m**, first add an import:

```
#import "DetailViewController.h"
```

› Then change the `didFinishLaunchingWithOptions` method to:

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
  self.window = [[UIWindow alloc] initWithFrame:
                                [[UIScreen mainScreen] bounds]];
  [self customizeAppearance];

  self.searchViewController = [[SearchViewController alloc]
           initWithNibName:@"SearchViewController" bundle:nil];

  if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
    self.splitViewController =
                          [[UISplitViewController alloc] init];

    DetailViewController *detailViewController =
        [[DetailViewController alloc]
            initWithNibName:@"DetailViewController" bundle:nil];

    UINavigationController *detailNavigationController =
      [[UINavigationController alloc]
               initWithRootViewController:detailViewController];

    self.splitViewController.delegate = detailViewController;
```

```
    self.splitViewController.viewControllers =
      @[self.searchViewController, detailNavigationController];

    self.window.rootViewController = self.splitViewController;
  } else {
    self.window.rootViewController = self.searchViewController;
  }

  [self.window makeKeyAndVisible];
  return YES;
}
```

This method has to figure out whether the app is running on the iPhone or on the iPad so it can decide whether to make the split-view controller.

You can check for iPad anywhere in your app with the following construct:

```
if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
    // running on an iPad
} else {
    // running on an iPhone or iPod touch
}
```

If the app runs on the iPhone, the app still does the same things as before: it creates the `SearchViewController` (in the `self.searchViewController` property) and makes it the `rootViewController` of the window.

On the iPad, however, the root view controller will be a `UISplitViewController`. You also create an instance of `DetailViewController` and put that inside a new `UINavigationController`. It is this navigation controller that you actually put in the split-view's detail pane. I'll tell you why you need this `UINavigationController` in a sec.

To place these view controllers into the split-view controller, you put them in an array and assign that to the `viewControllers` property. The first view controller from this array goes into the master pane, the second in the detail pane.

➤ In **DetailViewController.h**, change the interface line to conform to the `UISplitViewControllerDelegate` protocol:

```
@interface DetailViewController : UIViewController
                                  <UISplitViewControllerDelegate>
```

The app will build now but it won't work very well (try it out if you want to have a laugh). You really have to implement these split-view delegate methods first.

> **Note:** In this tutorial you have often added delegate protocols to the `@interface` line inside the .m file, i.e. to the private class extension. For example, the `DetailViewController` is also a `UIGestureRecognizerDelegate` but you wouldn't know that from looking at the .h file.
>
> The reason you're adding the `UISplitViewControllerDelegate` to the header file is that some other object – in this case the `AppDelegate` – needs to know that the `DetailViewController` can be a delegate for the split-view controller. But no other object cares about the fact that `DetailViewController` can handle gestures, so that delegate can stay hidden.

▸ In **DetailViewController.m**, add a new `masterPopoverController` property to the class extension:

```
@property (nonatomic, strong) UIPopoverController
                                        *masterPopoverController;
```

This property keeps track of the master pane in portrait orientation.

▸ Paste the following methods at the bottom:

```
#pragma mark - UISplitViewControllerDelegate

- (void)splitViewController:
                        (UISplitViewController *)splitController
   willHideViewController:(UIViewController *)viewController
   withBarButtonItem:(UIBarButtonItem *)barButtonItem
   forPopoverController:(UIPopoverController *)popoverController
{
  barButtonItem.title = NSLocalizedString(
                        @"Search", @"Split-view master button");
  [self.navigationItem setLeftBarButtonItem:barButtonItem
                                   animated:YES];
  self.masterPopoverController = popoverController;
}

- (void)splitViewController:
                        (UISplitViewController *)splitController
      willShowViewController:(UIViewController *)viewController
      invalidatingBarButtonItem:(UIBarButtonItem *)barButtonItem
{
  [self.navigationItem setLeftBarButtonItem:nil animated:YES];
  self.masterPopoverController = nil;
}
```

In portrait mode the master pane won't be visible all the time, only when you tap a button. This brings up a so-called **popover**. The split-view controller takes care of most of this logic for you but you still need to put that button somewhere.

That's why you put the `DetailViewController` inside a `UINavigationController` so you can put this button – which is a `UIBarButtonItem` – into its navigation bar. (You're not required to use a navigation controller for this; you could also add a toolbar to the `DetailViewController`, or use a different button altogether.)

Besides a `UIBarButtonItem`, the `willHideViewController` delegate method also gives you a `UIPopoverController` object, which controls the master pane. You will store a reference to this popover controller into the new `masterPopoverController` property, so you can refer to it later.

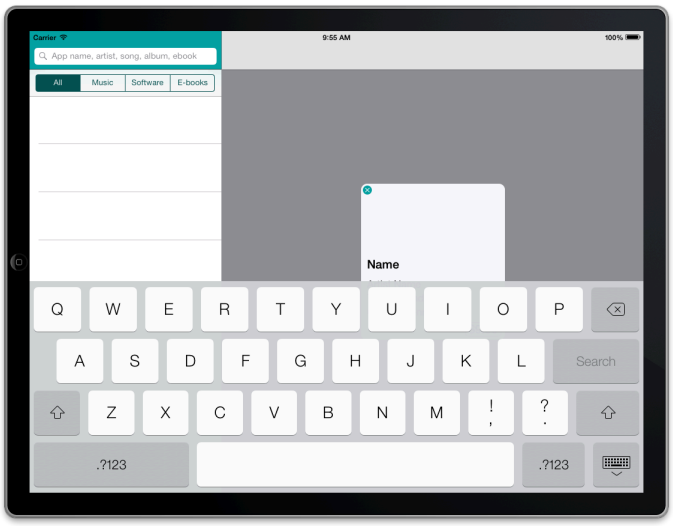
There is something else you should change before you can run the app. Currently the `SearchViewController` will present the `LandscapeViewController` when the device is rotated. That will seriously mess up what happens to the split-view controller so let's disable this feature for the iPad version.

➤ In **SearchViewController.m**, replace `willRotateToInterfaceOrientation` with:

```
- (void)willRotateToInterfaceOrientation:
                (UIInterfaceOrientation)toInterfaceOrientation
        duration:(NSTimeInterval)duration
{
  [super willRotateToInterfaceOrientation:toInterfaceOrientation
                                 duration:duration];

  if (UI_USER_INTERFACE_IDIOM() != UIUserInterfaceIdiomPad) {
    if (UIInterfaceOrientationIsPortrait(
                                      toInterfaceOrientation)) {
      [self hideLandscapeViewWithDuration:duration];
    } else {
      [self showLandscapeViewWithDuration:duration];
    }
  }
}
```

This will only show the `LandscapeViewController` when the app is not running on an iPad.

And that should be enough to get the app up and running with a split-view:



The app in a split-view controller

It will still take a bit of effort to make everything look good and work well, but this was the first step. If you play with the app you'll notice that it still uses the logic from the iPhone version and that doesn't always work so well now that the UI sits in a split-view.

For example, you can press the close button in the detail pane and the box from the `DetailViewController` will drop off the screen, never to return again (and the app will crash soon after). And if you tap a row in the search results then another `DetailViewController` pops up on top of the table view.

So in the rest of this section you'll be fixing up the app to make sure it doesn't do anything funny on the iPad!

› Add the following lines to **DetailViewController.m**'s `viewDidLoad` method:

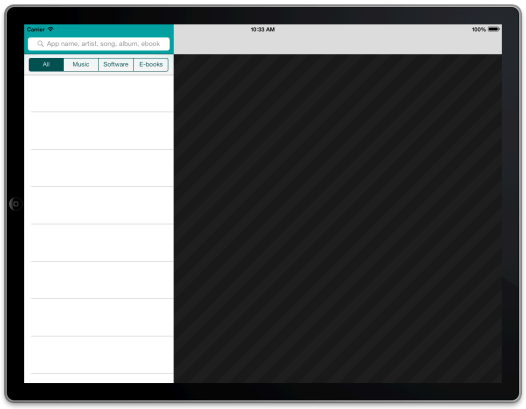
```
  if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {

    self.view.backgroundColor =
        [UIColor colorWithPatternImage:
                  [UIImage imageNamed:@"LandscapeBackground"]];

    self.closeButton.hidden = YES;

    self.popupView.hidden = (self.searchResult == nil);

  } else {
    UITapGestureRecognizer *gestureRecognizer =
          [[UITapGestureRecognizer alloc]
                  initWithTarget:self action:@selector(close:)];
    gestureRecognizer.cancelsTouchesInView = NO;
    gestureRecognizer.delegate = self;
```

```
    [self.view addGestureRecognizer:gestureRecognizer];

    self.view.backgroundColor = [UIColor clearColor];
  }
```

This always hides the close button so you cannot dismiss the `DetailViewController`, and it hides the labels when no `SearchResult` is selected in the main 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).

Note: You're supposed to move the code that adds the gesture recognizer into the else-clause, so that tapping the background has no effect on the iPad. Likewise for the line that sets the background color to `clearColor`, or the pattern image won't show up.

➤ Make an outlet for the close button.



Making the detail pane look a little better

Initially this means the `DetailViewController` doesn't show anything (except the patterned background), so you will need to change `SearchViewController` to tell the `DetailViewController` that a new `SearchResult` has been selected.

Previously, `SearchViewController` created a new instance of `DetailViewController` every time you tapped a row but now it will need to use the existing instance from the detail pane instead. But how does the `SearchViewController` know what that instance is?

You will have to give it a pointer to the `DetailViewController`; a good place for that is in `AppDelegate` where you create those instances.

➤ Add the following line to `didFinishLaunchingWithOptions`, inside the if-statement for the iPad:

```
self.searchViewController.detailViewController =
                                            detailViewController;
```

This won't work as-is because `SearchViewController` only has an instance variable named `_detailViewController`, not a property.

➤ Add a new property to **SearchViewController.h**'s public `@interface`:

```
@class DetailViewController;

@interface SearchViewController : UIViewController

@property (nonatomic, weak) DetailViewController
                                        *detailViewController;

@end
```

Of course, this requires a `@class` statement as well.

Notice that you make this property `weak`. The `SearchViewController` isn't responsible for keeping the `DetailViewController` alive (the split-view controller is) and the instance variable you used previously was `__weak` as well. By keeping the property weak, you don't have to do anything special to keep the same code working on the iPhone.

➤ In **SearchViewController.m**, remove the `_detailViewController` variable.

You need to make a few small changes because there is still some code that tries to access the old `_detailViewController` variable.

Exercise. Even though you just removed the `_detailViewController` variable, the app still builds without problems. Can you explain why? □

Answer: The property `detailViewController` has a backing instance variable that is also called `_detailViewController`. Because the old instance variable has the same name, it still works. However, when something is a property you really should access it through `self.propertyName`, not through its instance variable.

➤ Change the offending line in `showLandscapeViewWithDuration:` to:

```
[self.detailViewController
      dismissFromParentViewControllerWithAnimationType:
                          DetailViewControllerAnimationTypeFade];
```

➤ To change what happens when the user taps a search result on the iPad, replace `didSelectRowAtIndexPath` with:

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  [self.searchBar resignFirstResponder];
```

```
    SearchResult *searchResult =
                              _search.searchResults[indexPath.row];

    if (UI_USER_INTERFACE_IDIOM() != UIUserInterfaceIdiomPad) {
      [tableView deselectRowAtIndexPath:indexPath animated:YES];

      DetailViewController *controller =
           [[DetailViewController alloc]
              initWithNibName:@"DetailViewController" bundle:nil];

      controller.searchResult = searchResult;
      [controller presentInParentViewController:self];

      self.detailViewController = controller;
    } else {
      self.detailViewController.searchResult = searchResult;
    }
  }
```

On the iPhone this still does the same as before (pop up a new Detail screen), but on the iPad it simply assigns the existing `DetailViewController`'s `searchResult` property the new `SearchResult` object.

That by itself doesn't update the contents of the labels in the `DetailViewController`, so let's make that happen.

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

```
- (void)setSearchResult:(SearchResult *)newSearchResult
{
  if (_searchResult != newSearchResult) {
    _searchResult = newSearchResult;

    if ([self isViewLoaded]) {
      [self updateUI];
    }
  }
}
```

This is a so-called **setter** method and you've seen such methods a few times before. You can provide your own setter method when you want to perform certain functionality when a property changes.

First you have to put the new value into the property's backing instance variable, `_searchResult`. This is one of the few times that you need to access the instance variable of a property directly. You can't do `self.searchResult = newSearchResult` here, as that will call `setSearchResult:` again, which will do `self.searchResult =`

`...` again, which will call `setSearchResult:` again, to infinity and beyond. That is called an **infinite loop** and they're usually not a good idea.

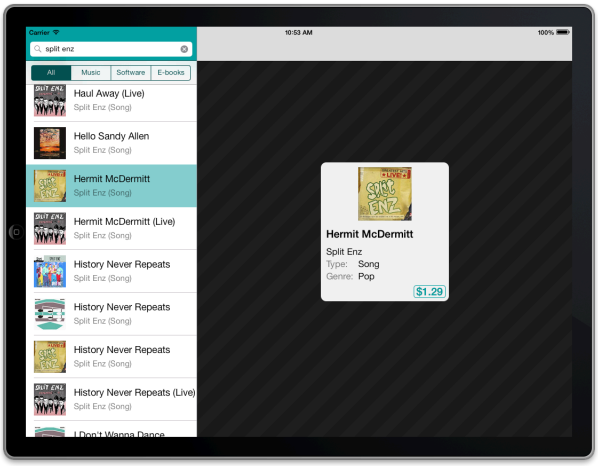
Once you've changed the value of the backing variable, you call the `updateUI` method to set the text on the labels. Notice that you first check whether the view is loaded. It's possible that `setSearchResult:` gets called when the `DetailViewController` hasn't loaded its view yet – which is exactly what happens in the iPhone version of the app –and you don't want to call `updateUI` then as there is no user interface yet to update at that point.

▸ Add the following lines to the bottom of `updateUI`:

```
if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
  self.popupView.hidden = NO;
}
```

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

You have successfully repurposed the Detail pop-up to also work in the detail pane of a split-view controller. Whether this is possible in your own apps depends on how different you want the user interfaces of the iPhone and iPad versions to be.

Often you'll find that the iPad user interface for your app is different enough from the iPhone's that you will have to make all new view controllers with some duplicated logic. If you're lucky you may be able to use the same view controllers for both versions of the app but often that is more trouble than it's worth.

More improvements

There are a bunch of small changes I'd like to make. For example, 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 **SearchViewController.m**'s `viewDidLoad` method, put the `becomeFirstResponder` call in an if-statement:

```
if (UI_USER_INTERFACE_IDIOM() != UIUserInterfaceIdiomPad) {
  [self.searchBar becomeFirstResponder];
}
```

The portrait mode needs some work also. When you tap on a row in the master pane, the contents of the detail pane change but the popover obscures about half of that. It would be better if you'd hide the popover after the user makes a selection. That's where the `masterPopoverController` property comes in handy.

▸ In **DetailViewController.m**'s `updateUI` method, change the last couple of lines:

```
if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
  self.popupView.hidden = NO;
  [self.masterPopoverController dismissPopoverAnimated:YES];
}
```

Now the master pane will fade out when you tap a row in the table.

It would be nice if the app shows its name in the big navigation bar on top of the detail pane. Currently all that space seems a bit wasted. Ideally, this would use the localized name of the app. You could use `NSLocalizedString()` and put the app name into the Localizable.strings files, but considering that you already put the localized app name in InfoPlist.strings it would be handy if you could use that. As it turns out, you can.

▸ Still in **DetailViewController.m**, add this line to the "UIUserInterfaceIdiomPad" section in `viewDidLoad`:

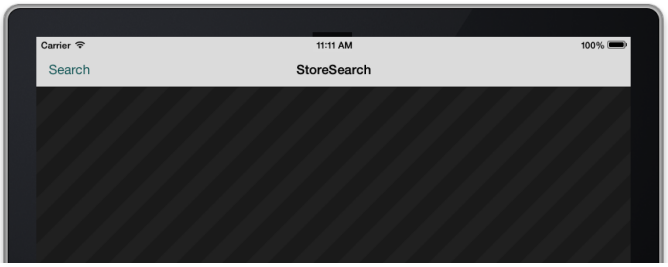
```
self.title = [[[NSBundle mainBundle] localizedInfoDictionary]
                            objectForKey:@"CFBundleDisplayName"];
```

The `self.title` property is used by the `UINavigationController` to put a title in the navigation bar. You set it to the value of the `CFBundleDisplayName` setting from the localized version of the Info.plist, in other words the translations from InfoPlist.strings.

If you were to run the app right now, no title would show up still (unless you still have the Simulator in Dutch) because you did not actually put a translation for `CFBundleDisplayName` in the English version of InfoPlist.strings.

➤ Add the following line to **InfoPlist.strings (English)**:

```
CFBundleDisplayName = "StoreSearch";
```



That's a good looking title

➤ 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 good to make sure. If you're satisfied everything still works as it should, then commit the changes.

Improving the detail pane

Even though you've placed the existing `DetailViewController` in the detail pane, the app is not using all that extra iPad space effectively. It would be good if you could keep using the same logic from the `DetailViewController` class but change the layout of its user interface.

You could do `if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad)` in `viewDidLoad` and then move all the controls around programmatically, but fortunately iOS allows you to simply load a different nib, depending on where the app is being used.

➤ In Finder, go to the **Base.lproj** folder and duplicate the file **DetailViewController.xib**. Rename the new file to **DetailViewController~ipad.xib**.

➤ Add **DetailViewController~ipad.xib** to the project and open it in Interface Builder.

➤ In the **Attributes inspector**, under **Simulated Metrics**, set **Size** to **Freeform**, and **Status Bar** to **None**. That will allow you to resize the main view.

➤ In the **Size inspector**, change the Width to 768 and Height to 1024.

➤ The iPad version doesn't use the **Close Button** so remove that button from the nib.

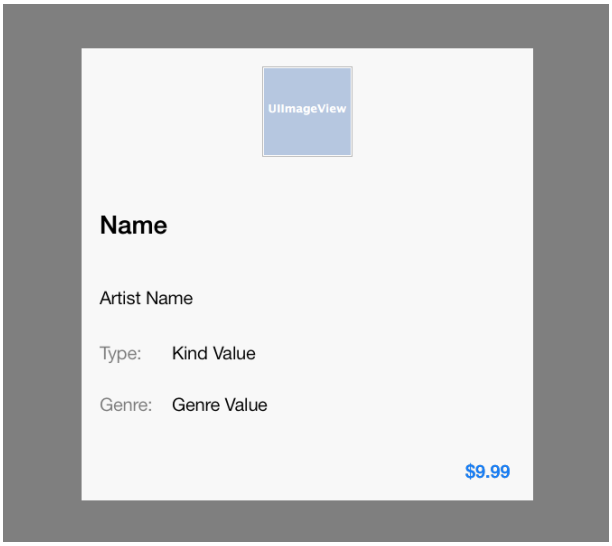
➤ Select the **Popup View** and change its size to 500 by 500 points. Place it in the center of the nib.

The alignment constraints for the Popup View should be blue now, but the size constraints are still orange. That's because the size of the view in the nib doesn't match the constraints that you have set on it.

➤ With the Popup View selected, open the **Resolve Auto Layout Issues** menu and choose **Update Constraints**. This resizes the Width and Height constraints to be 500 by 500 points as well. All the lines should be blue now.

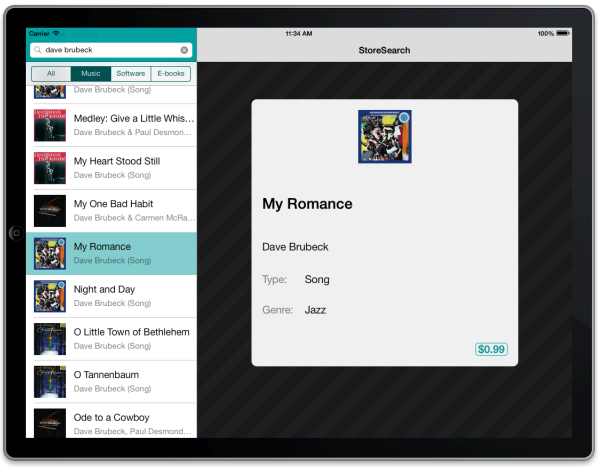
➤ Rearrange and resize the labels to take advantage of the extra space.

You should end up with something that looks like this:



Designing the new detail pane nib

And when you run the app:



The iPad now uses a different nib for the detail pane

Simply by adding `~ipad` to the filename you made the app load a different resource when it runs on the iPad.

By the way, you can get rid of the line in `viewDidLoad` that sets the `closeButton`'s `hidden` property, because the `closeButton` outlet will always be `nil` on the iPad.

➤ Add a Dutch localization for this nib. The only label you need to change is "Type:" to "Soort:".

Exercise. The first time the detail pane shows its contents is quite abrupt because you simply set the `hidden` property of `popupView` to `NO`, which causes it to instantaneously appear. See if you can make it show up using a cool animation. □

Your own popover

The split-view controller uses a popover for the master pane in portrait mode but you can also make your own popovers. They are a very handy UI element. A popover controller contains a "content view controller", which can be any view controller that you want. In this section you'll create a popover for a simple menu.

Let's start with the content view controller. This will be a regular `UITableViewController` with grouped style cells.

➤ Add a new file to the project. Name it **MenuViewController**, subclass of **UITableViewController**. Select **Targeted for iPad** but this time you will create no XIB, so *unselect* the **With XIB for user interface** option.

➤ In **MenuViewController.m**, replace the table view data source methods with:

```
- (NSInteger)numberOfSectionsInTableView:
                                    (UITableView *)tableView
{
  return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
            numberOfRowsInSection:(NSInteger)section
{
  return 3;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  static NSString *CellIdentifier = @"Cell";

  UITableViewCell *cell = [tableView
              dequeueReusableCellWithIdentifier:CellIdentifier];
  if (cell == nil) {
    cell = [[UITableViewCell alloc]
                        initWithStyle:UITableViewCellStyleDefault
```

```
                        reuseIdentifier:CellIdentifier];
  }

  if (indexPath.row == 0) {
    cell.textLabel.text = NSLocalizedString(
                @"Send Support Email", @"Menu: Email support");
  } else if (indexPath.row == 1) {
    cell.textLabel.text = NSLocalizedString(
                @"Rate this App", @"Menu: Rate app");
  } else {
    cell.textLabel.text = NSLocalizedString(
                @"About", @"Menu: About");
  }

  return cell;
}
```

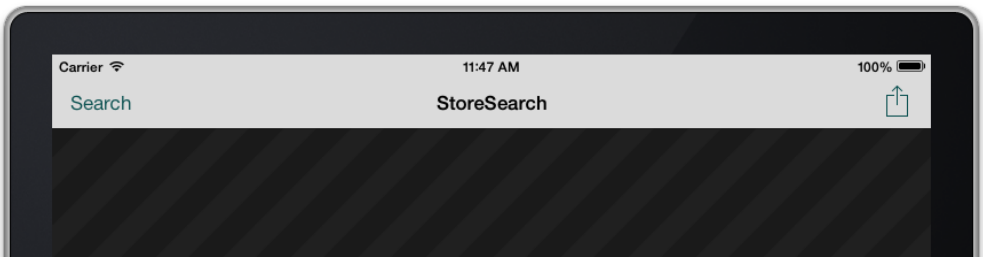
This just puts three items in the table. You will only do something with the first one in this tutorial. Feel free to implement the functionality of the other two by yourself.

To show this `MenuViewController` inside a popover, you first have to add a button to the navigation bar so that there is something to trigger the popover from. You can do this in **DetailViewController.m**'s `viewDidLoad` method.

➤ Add the following to `viewDidLoad` inside the if-statement for the iPad-specific controls:

```
self.navigationItem.rightBarButtonItem =
    [[UIBarButtonItem alloc]
         initWithBarButtonSystemItem:UIBarButtonSystemItemAction
         target:self action:@selector(menuButtonPressed:)];
```

This creates a new `UIBarButtonItem` object and puts it on the right-hand side of the navigation bar, like this:



A new button for the menu in the navigation bar

➤ First, import the new class at the top of **DetailViewController.m**:

```
#import "MenuViewController.h"
```

➤ Add a property for the new popover controller:

```
@property (nonatomic, strong) UIPopoverController
                                          *menuPopoverController;
```

➤ And add the `menuButtonPressed:` method to show this popover:

```
- (void)menuButtonPressed:(UIBarButtonItem *)sender
{
  [self.menuPopoverController
        presentPopoverFromBarButtonItem:sender
        permittedArrowDirections:UIPopoverArrowDirectionAny
        animated:YES];
}
```

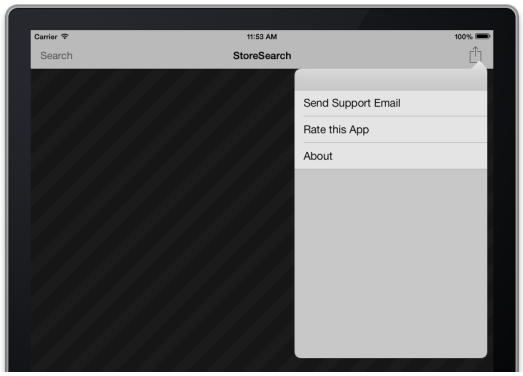
That's all the code you need to present a popover controller. The creation of the popover object is done lazily by the `self.menuPopoverController` getter.

➤ Implement the getter method:

```
- (UIPopoverController *)menuPopoverController
{
  if (_menuPopoverController == nil) {
    MenuViewController *menuViewController =
              [[MenuViewController alloc]
                          initWithStyle:UITableViewStyleGrouped];

    _menuPopoverController = [[UIPopoverController alloc]
               initWithContentViewController:menuViewController];
  }
  return _menuPopoverController;
}
```

This uses lazy loading to delay the allocation of the `MenuViewController` and `UIPopoverController` objects until the popover is actually needed.

If you run the app and press the menu button, the app looks like this:



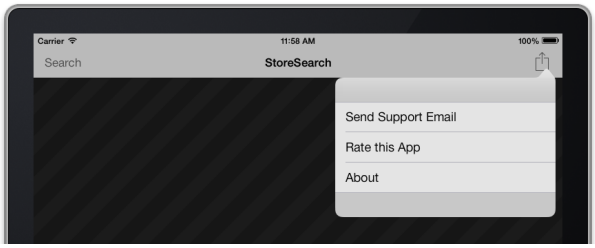
That menu is a bit too tall

The popover controller 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 `MenuViewController` should be with the `preferredContentSize` property.

▸ Change **MenuViewController.m**'s `viewDidLoad` to:

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  self.preferredContentSize = CGSizeMake(320, 202);
}
```

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 he can use the rest of the screen again (you can make exceptions to this by setting the popover controller's `passthroughViews` property).

The only thing that still does respond to touches is the `UIBarButtonItem`. This is done so you can tap the button again to dismiss the popover. Let's add that functionality to this app as well.

▸ Change the `menuButtonPressed:` method in `DetailViewController` to the following:

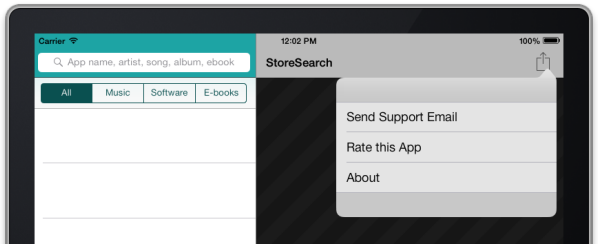
```
- (void)menuButtonPressed:(UIBarButtonItem *)sender
```

```
{
  if ([self.menuPopoverController isPopoverVisible]) {
    [self.menuPopoverController dismissPopoverAnimated:YES];
  } else {
    [self.menuPopoverController
            presentPopoverFromBarButtonItem:sender
            permittedArrowDirections:UIPopoverArrowDirectionAny
            animated:YES];
  }
}
```

This first checks if the popover is visible. If it is, then you dismiss it with an animation.

➤ Try it out. Tapping the bar button item again while the popover is visible will close it.

While the menu popover is visible, the other bar button (Search) is still active as well, if you're in portrait mode. This can create a situation where two popovers are open at the same time:



Both popovers are visible

That is a violation of the rules from Apple's Human Interface Guidelines, also known as the "HIG". The folks at Apple don't like it when apps show more than one popover at a time, probably because it is confusing to the user which one requires input. The app will be rejected from the App Store for this, so you have to make sure this situation cannot happen.

The scenario you need to handle is when the user first opens the menu popover followed by a tap on the Search button. To fix this issue, you need to know when the Search button is pressed and the master pane becomes visible, so you can hide the menu popover. Wouldn't you know it… of course there is a delegate method for that.

➤ Add the following method to the section that holds the split-view delegate methods:

```
- (void)splitViewController:
                        (UISplitViewController *)splitController
    popoverController:(UIPopoverController *)popoverController
```

```
    willPresentViewController:(UIViewController *)viewController
{
  if ([self.menuPopoverController isPopoverVisible]) {
    [self.menuPopoverController dismissPopoverAnimated:YES];
  }
}
```

And that should do it!

Private APIs

Sometimes you'll hear developers refer to using "private libraries" or "private APIs" in their apps. These developers are using methods and properties that are not officially recognized by Apple – functionality that is hidden in the frameworks and thus not part of the public API.

You're not supposed to use such private APIs for two reasons: 1) these APIs may have unexpected side effects and not be as robust as their publicly available relatives; 2) these APIs may suddenly disappear in future versions of the OS. Sometimes, however, using a private API is the only way to access certain functionality on the device. If so, you're out of luck.

Even though Apple doesn't want you to use these private APIs, some really persistent developers will find ways to use them. If you do, there is a big chance that your app is rejected from the App Store as Apple scans apps for the use of these APIs. Worse, your app may suddenly stop working. Fortunately, for most apps the official public APIs are more than enough and you won't need to resort to the private stuff.

Sending email from within the app

Now let's make the "Send Support Email" menu option work. Letting users send an email from within your app is pretty easy. iOS provides the `MFMailComposeViewController` class that will take care of everything for you. It lets the user type the email and then sends the email using the mail account that the user set up on the device.

All you have to do is create an `MFMailComposeViewController` object and present it on the screen.

➤ In **MenuViewController.m**, add an import:

```
#import "DetailViewController.h"
```

➤ Add the `didSelectRowAtIndexPath` method so that it will handle a tap on the Send Support Email row:

```
- (void)tableView:(UITableView *)tableView
        didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  [tableView deselectRowAtIndexPath:indexPath animated:YES];

  if (indexPath.row == 0) {
    [self.detailViewController sendSupportEmail];
  }
}
```

You will let the `DetailViewController` handle the sending of the email, mainly because this lets you dismiss the popover controller when this menu option is chosen and `DetailViewController` is the only object that knows anything about the popover controller. To make this work, `MenuViewController` needs to get a property that references `DetailViewController`.

➤ Add that property to the public `@interface` of **MenuViewController.h**:

```
@class DetailViewController;

@interface MenuViewController : UITableViewController

@property (nonatomic, weak) DetailViewController
                                        *detailViewController;

@end
```

Of course, you have to put something into this property for all of this to work.

➤ In **DetailViewController.m**, change the getter for `menuPopoverController` to the following:

```
- (UIPopoverController *)menuPopoverController
{
  if (_menuPopoverController == nil) {
    MenuViewController *menuViewController =
              [[MenuViewController alloc]
                        initWithStyle:UITableViewStyleGrouped];

    menuViewController.detailViewController = self;

    _menuPopoverController = [[UIPopoverController alloc]
              initWithContentViewController:menuViewController];
  }
  return _menuPopoverController;
}
```

This tells the `menuViewController` object who the `DetailViewController` is. You could have used a delegate to make this connection but in this case I think the connection between these two classes is so strong that you don't need to abstract it with a delegate protocol.

➤ Add the method signature for `sendSupportEmail` to the public interface of **DetailViewController.h**:

```
- (void)sendSupportEmail;
```

➤ The `MFMailComposeViewController` lives in the MessageUI framework, so import that in **DetailViewController.m**:

```
#import <MessageUI/MessageUI.h>
```

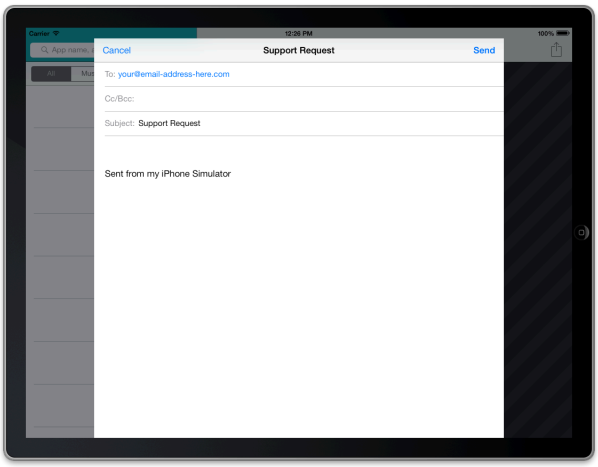
➤ And put the actual method in its implementation:

```
- (void)sendSupportEmail
{
  [self.menuPopoverController dismissPopoverAnimated:YES];

  MFMailComposeViewController *controller =
                    [[MFMailComposeViewController alloc] init];

  if (controller != nil) {
    [controller setSubject:NSLocalizedString(
                         @"Support Request", @"Email subject")];
    [controller setToRecipients:
                             @[@"your@email-address-here.com"]];

    [self presentViewController:controller animated:YES
                    completion:nil];
  }
}
```

You create an `MFMailComposeViewController` object and set some of its properties, the subject of the email and the email address of the recipient. You probably should put your own email address here! Finally, it shows the new view controller on the screen.

➤ The app must also link with the MessageUI framework. To do this, go to the **Project Settings** screen and add **MessageUI.framework** under **Linked Frameworks and Libraries**.

➤ Run the app and pick the Send Support Email menu option. A form slides up the screen that lets you write an email.



The email interface

Notice that the Send and Cancel buttons don't actually appear to do anything. That's because you still need to implement the delegate for this screen.

▸ Add a new delegate protocol to **DetailViewController.m**'s `@interface` line:

```
@interface DetailViewController () <UIGestureRecognizerDelegate,
                                    MFMailComposeViewControllerDelegate>
```

▸ In the `sendSupportEmail` method, add the following line:

```
controller.mailComposeDelegate = self;
```

▸ And finally, add the delegate method at the bottom of the file:

```
#pragma mark - MFMailComposeViewControllerDelegate

- (void)mailComposeController:
                  (MFMailComposeViewController *)controller
    didFinishWithResult:(MFMailComposeResult)result
                  error:(NSError *)error
{
  [self dismissViewControllerAnimated:YES completion:nil];
}
```

The `result` parameter says whether the mail could be successfully sent or not. This app doesn't really care about that, but you could show an alert in case of an error if you want. Check the documentation for the possible result codes.

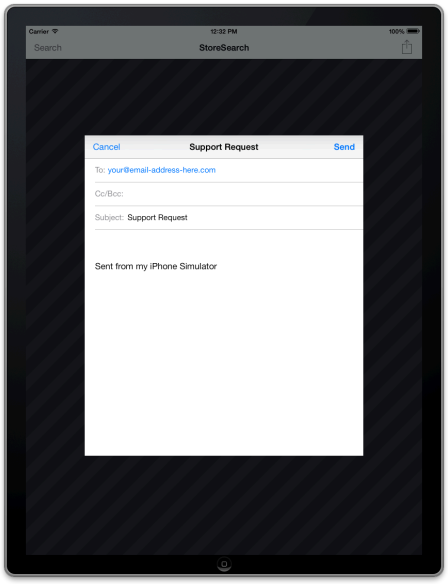
▸ Now if you press Cancel or Send, the mail compose sheet gets dismissed. If you're testing on the Simulator, no email actually gets sent out, so don't worry about spamming anyone when you're testing this.

Did you notice that the mail form did not take up the entire space in the screen in landscape, but when you rotate to portrait it does? That is called a **page sheet**. On the iPhone if you presented a modal view controller it always took over the entire screen, but on the iPad you have several options. The page sheet is probably the nicest option for the `MFMailComposeViewController`, but let's experiment with the other ones as well, shall we?

▸ In `sendSupportEmail`, add the following line:

```
controller.modalPresentationStyle =
                                UIModalPresentationFormSheet;
```

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:



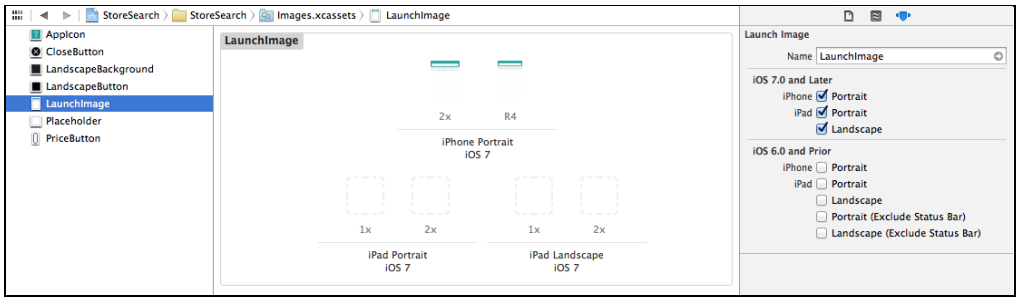
The email interface in a form sheet

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!

Launch images

The iPad version of the app now looks a lot different than the iPhone version, so it really needs its own set of launch images.

▸ Open the asset catalog, **Images.xcassets** and select the **LaunchImage** group. In the **Attributes inspector** put checkmarks next to the **iPad Portrait** and **Landscape** options:



Adding the iPad launch images to the asset catalog

The **Launch Images iPad** folder from this chapter's resources contains four images: **Default-Portrait~ipad.png**, **Default-Landscape~ipad.png**, and their **@2x** counterparts. (Note that unlike the iPhone, the iPad still needs low-resolution graphics, because iOS 7 also runs on non-Retina iPads.)

➤ Drag these images into the corresponding slots in the asset catalog.

And that's it for the StoreSearch app! Congratulations for making it this far, it has been a long tutorial.

➤ Celebrate by committing the final version of the source code and tagging it v1.0!

You can find the project files for the complete app under **12 - Finished App** in the tutorial's Source Code folder.

What do you do with an app that is finished? Upload it to the App Store, of course! (And with a little luck, make some big bucks...)

Distributing the app

Throughout these tutorials you've probably been testing the apps on the Simulator and occasionally on your device. That's great, but when the app is nearly done you may want to let other people beta test it. You can do this on iOS with so-called **ad hoc** distributions.

Your Developer Program membership allows you to register up to 100 devices with your account and to distribute your apps to the users of those devices, without requiring that they buy the apps from the App Store. You simply build your app in Xcode and send them a ZIP file that contains your application bundle and your Ad Hoc Distribution profile. Your beta testers can then drag these files into iTunes and sync their iPhones and iPads to install the app.

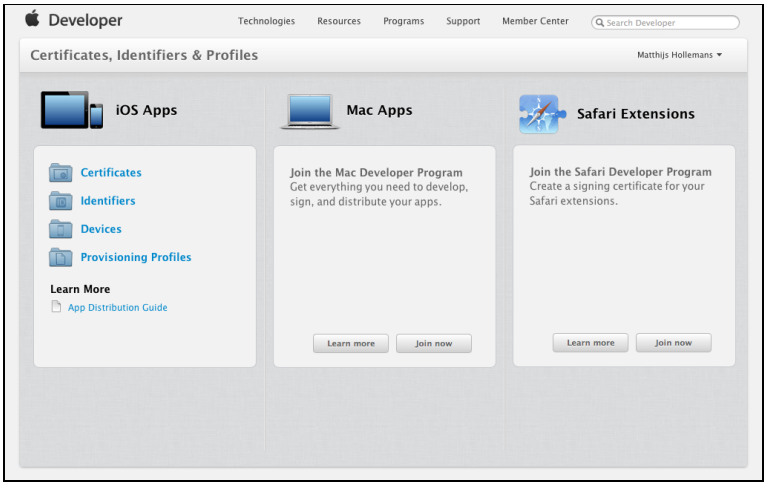
In this section you'll learn how to make an Ad Hoc distribution for the StoreSearch app. Later on I'll also show you how to submit the app to the App Store, which is a very similar process. (By the way, I'd appreciate it if you don't actually submit the apps from these tutorials. I don't want to spam the App Store with dozens of identical "StoreSearch" or "Bull's Eye" apps.)

The distribution profile

Before you can put your app on a device, it must be signed with your **certificate** and a **profile**. So far when you've run apps on your device, you have used the Team Provisioning Profile but that is for development purposes only and can only be used from within Xcode.

You probably don't want to send your app's source code to your beta testers, or require them to mess around with Xcode, so you must create a new profile that is just for distribution.

› Open your favorite web browser and surf to the iOS Dev Center at http://developer.apple.com/ios/. Sign in and go to **Certificates, Identifiers & Profiles** in the right sidebar.

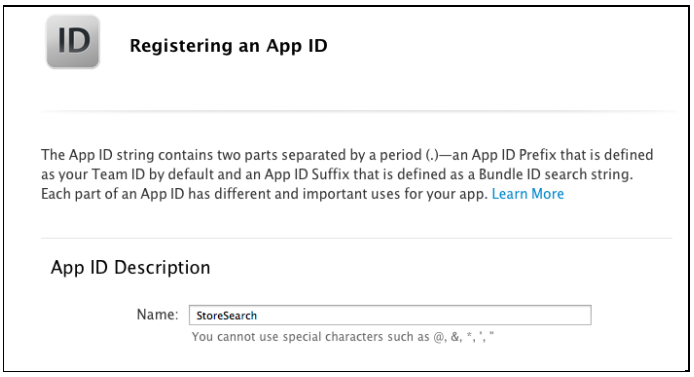


The Certificates, Identifiers & Profiles section of the iOS Dev Center

Note: Like any piece of software, the iOS Dev Center changes every now and then. It's possible that by the time you read this, some of the items are in different places or have different names. The general flow should still be the same, though. And if you really get stuck, online help is usually available.

Tip: If using this website gives you problems such as pages not loading correctly, then try it with Safari. Other browsers sometimes give strange errors.

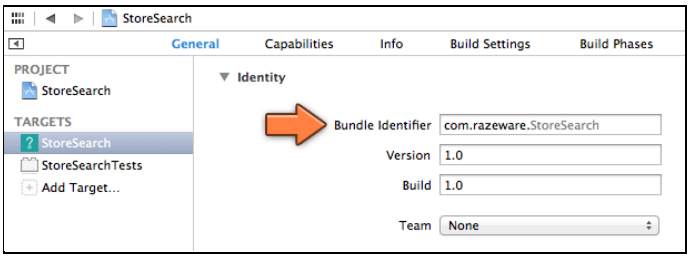
› Click on **Identifiers** (under iOS Apps). In the new page that appears, press the **+** button to add a new App ID:



Creating a new App ID

› Fill in the **App ID Description** field. This can be anything you want – it's just for usage on the Provisioning Portal.

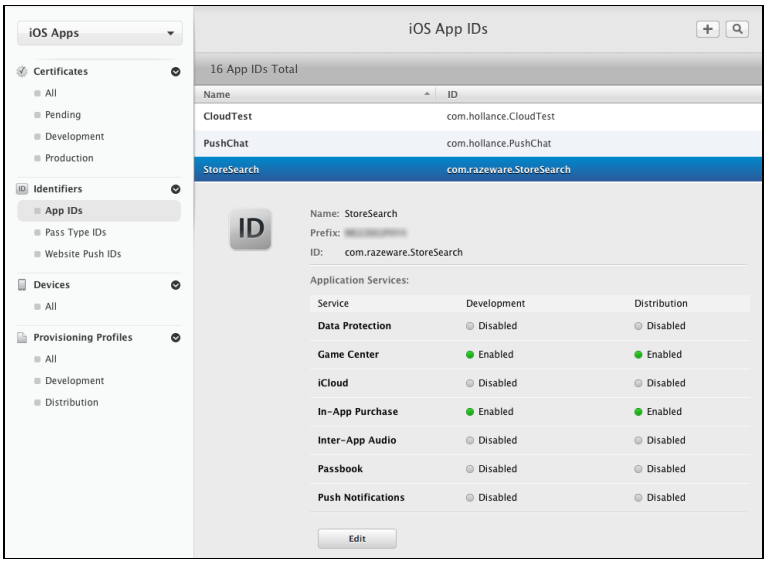
› Under **App ID Suffix**, select **Explicit App ID**. In the **Bundle ID** field you must enter the identifier that you used when you created the Xcode project. For me that is **com.razeware.StoreSearch**.



The Bundle ID must match with the identifier from Xcode

› Leave the other fields as they are and press **Continue** to create the App ID.

The portal will now generate the App ID for you and add it to the list:



The new App ID

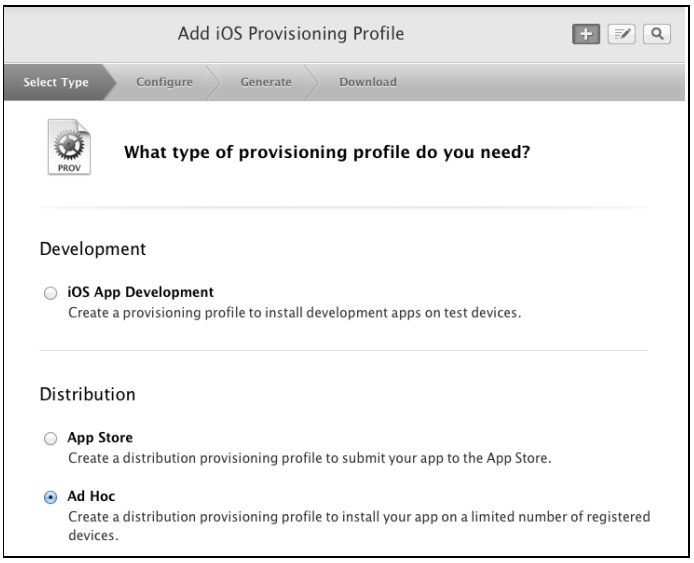
If you want your app to support push notifications, In-App Purchases, or iCloud then you can configure that here. StoreSearch doesn't need any of that so leave it as-is.

Notice that the full App ID is something like **U89ECKP4Y4.com.yourname.StoreSearch**. That number in front is known as the "bundle seed" but you don't need to worry about that.

➤ In the left-hand menu, under Provisioning Profiles, click **Distribution**. This will show your current distribution profiles. (You probably don't have any yet.)

There are two types of distribution profiles: Ad Hoc and App Store. You'll first make an Ad Hoc profile.

➤ Click the + button to create a new profile:



Creating a new provisioning profile for distribution

› Select **Ad Hoc** and click **Continue**.

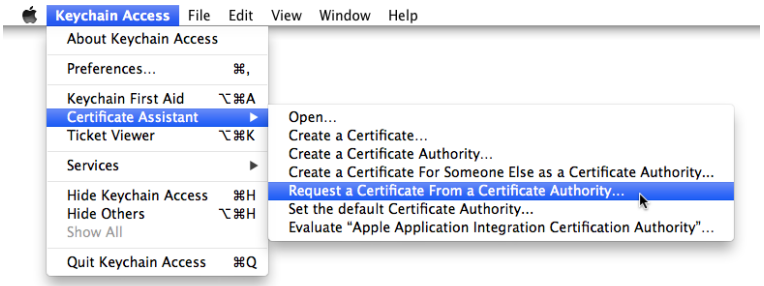
› The next step asks you to select an App ID. Pick the App ID that you just created ("StoreSearch").

› If you do not have a distribution certificate yet, the portal asks you to create one. Click the **Create Certificate** button. Select the **App Store and Ad Hoc** type.

As part of the certificate creation process you need to generate a CSR or Certificate Signing Request. It sounds scary but follow these steps and you'll be fine:

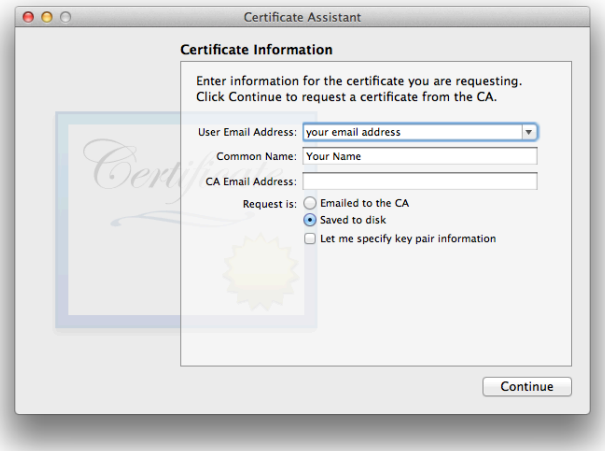
› Open the **Keychain Access** app on your Mac (it is in Applications/Utilities).

› From the **Keychain Access** menu, choose **Certificate Assistant → Request a Certificate from a Certificate Authority...**:



Using Keychain Access to create a CSR

› Fill out the fields in the window that pops up:



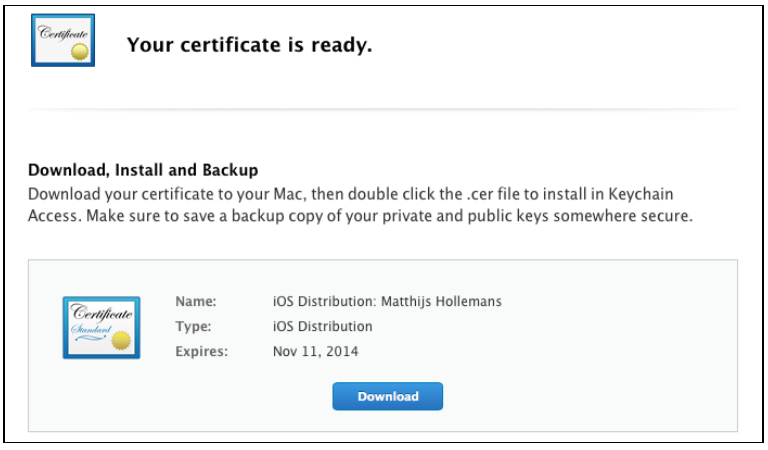
Filling out the certificate information

- **User Email Address**: Enter the email address that you use to sign into the iOS Dev Center. In other words, this is the Apple ID from your Developer Program account.
- **Common Name**: Fill in your name or your company's name.

› Check the **Saved to disk** option and press **Continue**. Save the file to your Desktop.

› Go back to the web browser and go to the next step. Upload the **CertificateSigningRequest.certSigningRequest** file you just created and click **Generate**.

After a couple of seconds you should be the owner of a brand new distribution certificate.



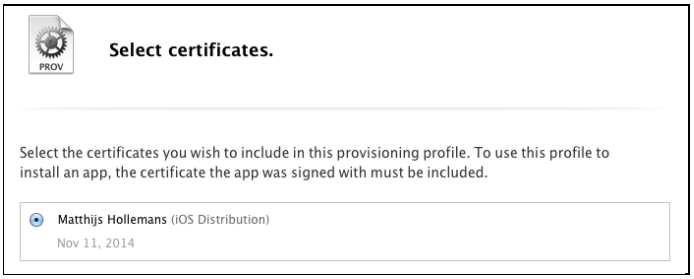
The certificate was successfully created

› Click the big **Download** button. This downloads a file named **ios_distribution.cer**. Double-click this file to install it.

You don't get any confirmation from this but you should be able to see the new certificate in the Keychain Access app under My Certificates.

➤ Go back to the portal and click on **Provisioning Profiles**, **Distribution** again. Unfortunately, you will have to repeat some of the steps. Click the **+** button, choose **Ad Hoc**, and choose the App ID.

Now the portal asks you to select the certificate that should be used to create this provisioning profile:

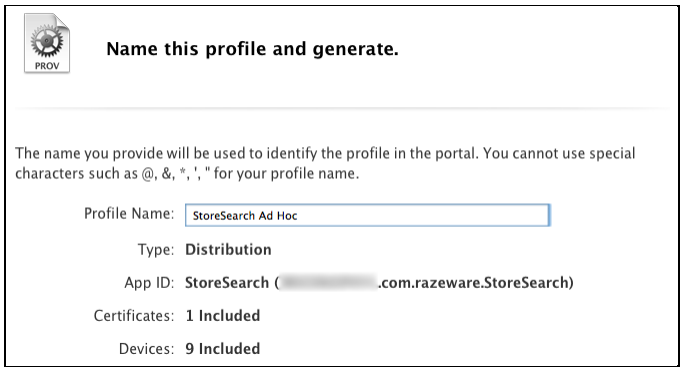


Selecting the certificate

➤ Select the certificate and click **Continue**.

In the next step you need to select the devices for which the provisioning profile is valid. If you're sending the app to beta testers, their devices need to be included in this list. (To add a new device, use the Devices menu option in the portal).

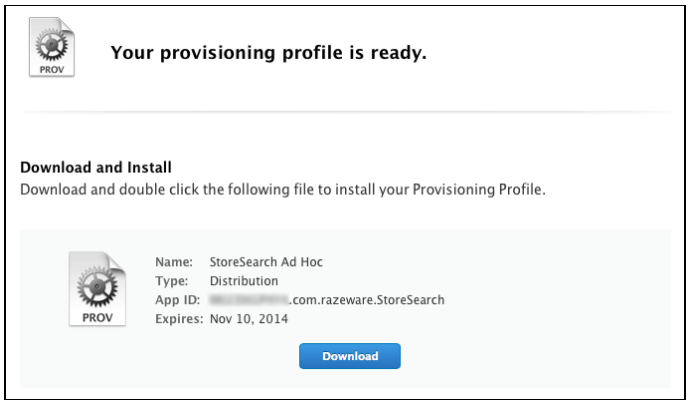
➤ Select your device(s) from the list and click **Continue**.



➤ Give the profile a name, for example **StoreSearch Ad Hoc**. Picking a good name is useful for when you have a lot of apps.

➤ If everything looks OK, click **Generate**.

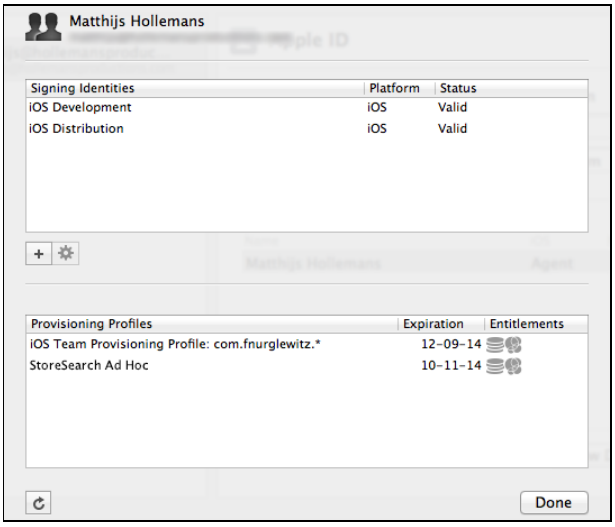
After a few seconds the provisioning profile is ready for download.



The provisioning profile was successfully created

› Click the **Download** button. This downloads the file **StoreSearch_Ad_Hoc.mobileprovision**. Double-click to install it. (You can also drag it on top of the Xcode icon in your dock.)

› Go back to Xcode and open the **Preferences** window. Go to the **Accounts** tab and click on **View Details...** You should see something like this:



The certificates and provisioning profiles for this account

There are two "signing identities" (i.e. certificates) in the list: the development certificate that you made way back in the Bull's Eye chapter and the distribution certificate that you made just now. There are also two provisioning profiles: the Team profile that you've been using to test the apps on your device, and the new **StoreSearch Ad Hoc** profile.

Great, you're just about ready to build the app for distribution.

Debug builds vs. Release builds

Xcode can build your app in a variety of **build configurations**. Projects come standard with two build configurations, Debug and Release. While you were developing the app you've always been using Debug mode, but when you build your app for distribution it will use Release mode.

The difference is that in Release mode, certain optimizations will be turned on to make your code as fast as possible, and certain debugging tools will be turned off. Not including the debugging tools will make your code smaller and faster – they're not much use to an end user anyway.

However, changing how your app gets built does mean that your app may act differently under certain circumstances. Therefore it's a good idea to give your app a thorough testing in Release mode as well, preferably by doing an Ad Hoc install on your own devices. That is the closest you will get to simulating what a user sees when he downloads your app from the App Store.

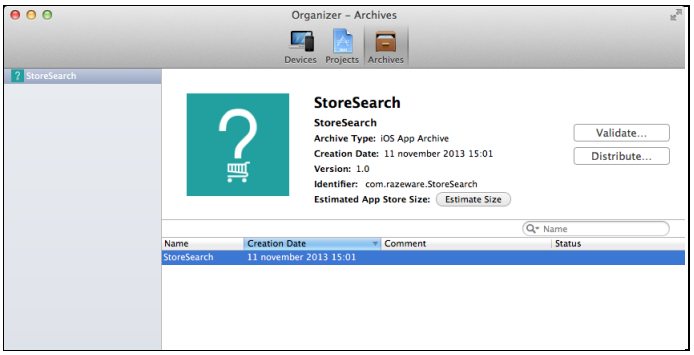
You can add additional build configurations if you want. Some people add a new configuration for Ad Hoc and another for App Store that lets them tweak the build settings for the different types of distribution.

➤ First, in the scheme picker at the top of the Xcode window choose **iOS Device** (or the name of your device if it is connected to your Mac) rather than the Simulator.

➤ From the Xcode menu bar, select **Product** → **Archive**. If the Archive option is grayed out, then the scheme is probably set to Simulator rather than the device.

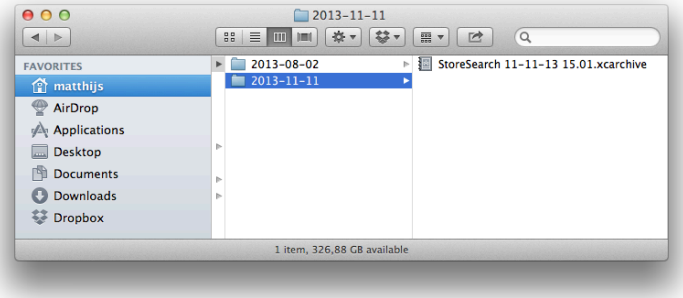
Now Xcode will build the app. By default, the Archive operation uses the Release build configuration.

➤ When the build is done and without errors, Xcode opens the Organizer window on the Archives tab:



The Archives tab in the Xcode Organizer

If you right-click the archive in the list and choose **Show in Finder**, the folder that contains the archive file opens:



The archive in Finder

If you right-click the **.xcarchive** file and choose **Show Package Contents**, you can take a peek inside. In the folder Products you will find the application bundle. To see what is in the application bundle, right-click it and choose Show Package Contents again.

dSYM files

The folder **dSYMs** inside the archive contains a very important file named **StoreSearch.app.dSYM**. This dSYM file contains symbolic names for the classes and methods in your app. That information has been stripped out from the final executable but is of vital importance if you receive a crash report from a customer. (You can download these crash reports through the iTunes Connect website.)

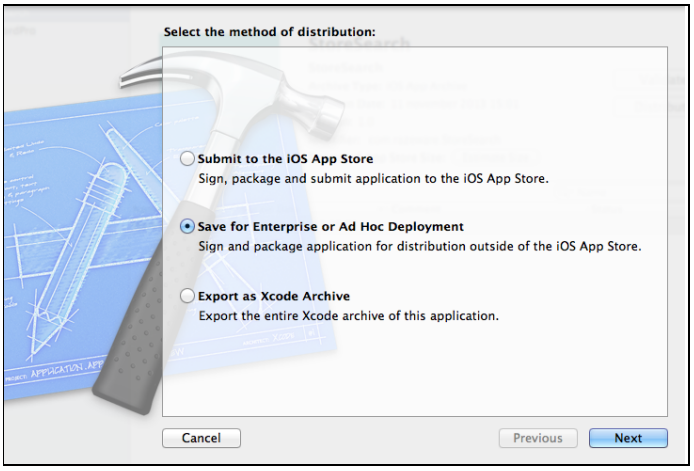
Crash reports just contains a bunch of numbers that are meaningless unless combined with the debug symbols from the dSYM file. When properly "symbolicated", the crash log will tell you where the crash happened – which is essential for debugging! – but in order for that to work Xcode must be able to find the dSYM files.

So it is important that you don't throw away these .xcarchive files for the versions of your app that you send to beta testers or the App Store. You don't have to keep them in the folder where Xcode puts them per se, but you should keep them around somewhere and back them up.

You don't want to get crash reports that you can't make any sense of! (Of course, the best situation is that you don't get any crash reports at all!)

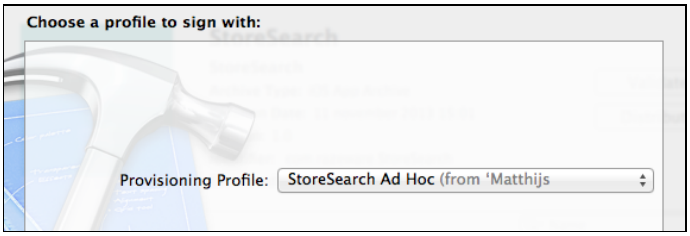
The .xcarchive isn't the thing that you will actually send to your beta testers. Instead, Xcode will build another package that is based on the contents of this archive.

› Select the archive from the list and press the **Distribute** button. In the screen that appears, select the **Save for Enterprise or Ad Hoc Deployment** option. Click **Next**.



Choosing the method of distribution

Now Xcode asks which provisioning profile to sign the package with:



Choosing the provisioning profile

› Select the **StoreSearch Ad Hoc** profile and press **Export**.

You may get a message that says **codesign wants to sign using key "Your Name" in your keychain**. This is Xcode asking for permission to use your distribution certificate. Click the **Always Allow** button or it will ask every time, which gets annoying quickly.

› Choose the location to save the file (I usually pick the Desktop). Name it **StoreSearch.ipa**. This is the file that you will give to your beta testers. (Make sure the "Save for Enterprise Distribution" option is disabled.)

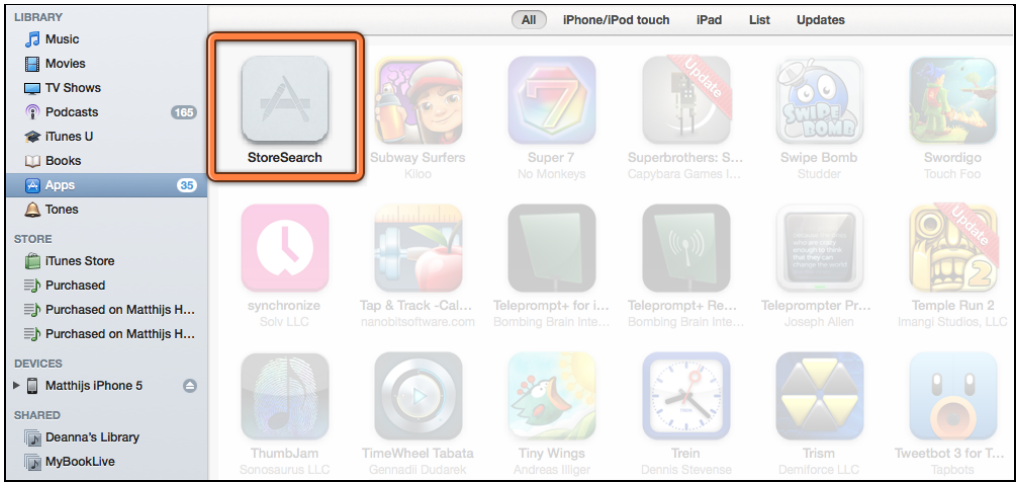
The .ipa file

An IPA file is simply a ZIP file that contains a folder named "Payload" and your application bundle. Give this **.ipa** together with **StoreSearch_Ad_Hoc.mobileprovision** to your beta testers and they will be able to run the app on their devices.

This is what they have to do. It's probably a good idea if you follow along with these steps, so you can verify that the Ad Hoc build actually worked.

1. Open iTunes and go to the Apps screen.
2. Drag **StoreSearch.ipa** into the Apps screen.
3. Drag **StoreSearch_Ad_Hoc.mobileprovision** file into the Apps screen.
4. Connect your iPhone or iPad to the computer.
5. Sync with iTunes.

That's it. Now the app should appear on the device. If iTunes balks and gives an error, then nine times out of ten you did not sign the app with the correct Ad Hoc profile or the user's device ID is not registered with the profile.



StoreSearch in the Apps section of iTunes

Ad Hoc distribution is pretty handy. You can send versions of the app to beta testers (or clients if you are into contract development) without having to upload the app to the App Store first.

There are practical limits to Ad Hoc distribution, primarily because it is intended as a testing mechanism, not as an alternative to the App Store. For example, Ad Hoc profiles are valid only for a couple of months (currently four), and you can only register 100 devices. Note that you can reset these device IDs only once per year so be judicious about registering new devices.

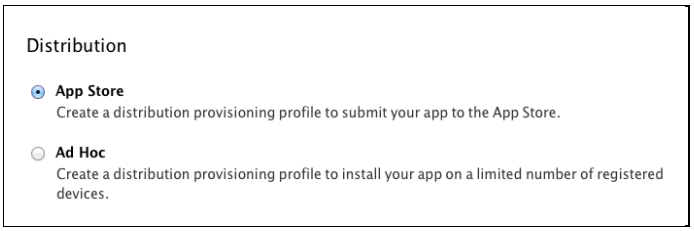
It's a good idea to test your apps using Ad Hoc distribution before you submit them to the App Store, just so you're sure everything works as it's supposed to outside of Xcode.

Note: If you're serious about beta testing your apps – and you should be – then look into using a service such as TestFlight (https://www.testflightapp.com) or HockeyApp (http://hockeyapp.net). With those services your testers don't need to fuss with IPA files and iTunes. They can simply install the app on their devices by surfing to a special URL in Mobile Safari. It doesn't get much easier than that!

Submitting to the App Store

After months of slaving away at your new app, version 1.0 is finally ready. Now all that remains is submitting it to the App Store. Doing so is actually fairly straightforward and I'll show you the steps here.

- You will need to create a new distribution profile on the iOS Dev Center first. This time you'll make an **App Store** profile.

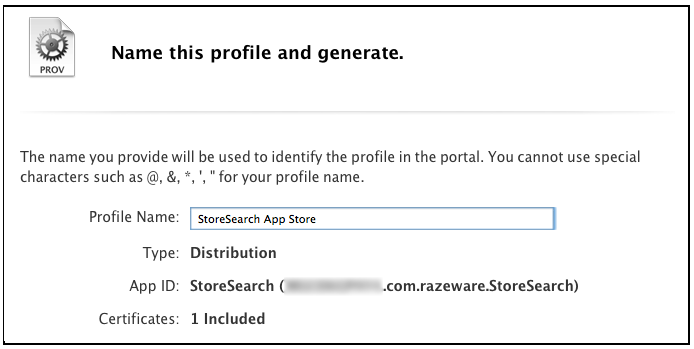


Choosing the App Store distribution profile

- The next step asks you for the App ID. Select the same App ID as before.

- The third step asks for your distribution certificate. Select the same certificate as before.

There is no step for choosing devices; that is only required for Ad Hoc distribution.



Naming the provisioning profile

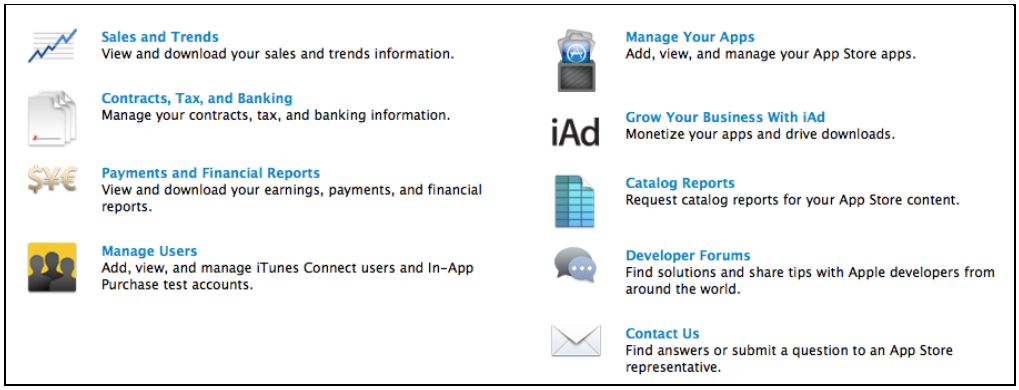
- Give the profile the name **StoreSearch App Store** and click **Generate**.

- Download the profile and double click the **StoreSearch_App_Store.mobileprovision** file to install it. Verify that the profile was installed in the Xcode Preferences, under Accounts.

You do not have to re-build the app. You can use the archived version that you made earlier (because no doubt you have tested the Ad Hoc version and found no bugs). However, you first have to set up the application on iTunes Connect.

› Surf to https://itunesconnect.apple.com. Sign in using your Developer Program account.

If you've never been here before, then make sure to visit the **Contracts, Tax, and Banking** section and fill out the forms. All that stuff has to be in order before your app can be sold on the store.



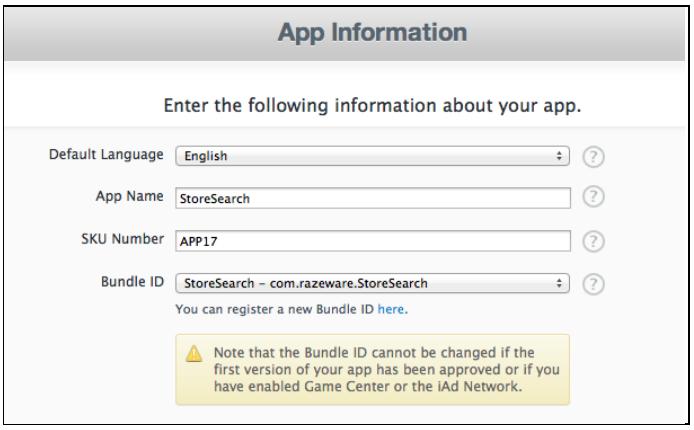
The iTunes Connect web site

> **Tip:** It is best to use Safari to visit iTunes Connect. This website doesn't always work so well on other browsers.

› When you've taken care of the administrativia, click on **Manage Your Apps** and then the **Add New App** button.

› If this is the first app that you're adding, you will be asked to enter the name under which you wish to publish your apps on the store. You can use your own name or a company name, but choose carefully, you only get to pick this once and it's a big hassle to change later! (Believe it or not, Apple requires you make such changes by fax. That's right, by fax.)

› Now is the time to enter some basic details about the app:



Entering the name and bundle ID of the app

I entered **StoreSearch** as the name for the app. The SKU is an identifier for your own use; it stands for "stock-keeping unit". When you get sales reports, they include this SKU. It's purely something for your own administration. For Bundle ID you pick the App ID that you used to make the distribution provisioning profile.

> **Note:** The iTunes Connect interface changes from time to time, so what you see in your browser may be slightly different. Don't worry about the details; the basic idea is still the same.

In the next couple of steps you are asked:

- When your app should become available
- What the price will be
- The version number
- A description that will be visible on the store
- The primary and secondary category that it will be listed under
- A list of keywords that customers can search for (limited to 100 characters)
- A URL to your website and support pages
- Contact information. Apple will contact you at this address if there are any problems with your submission.
- Notes for the reviewer. These are optional but a good idea if your app requires a login of some kind. The reviewer will need to be able to login to your app or service in order to test it.
- A rating if your app contains potentially offensive material
- A 1024×1024 icon image
- Up to five screenshots per device. You need to supply screenshots for 3.5-inch Retina screens, 4-inch Retina screens, and iPad.

It's a good idea to prepare this information beforehand. You can read the iTunes Connect Developer Guide for more info (available from the home page). If your app supports multiple languages, then you can also supply a translated description, screenshots and even application name.

Make a good first impression

People who are searching or browsing the store for cool new apps generally look at things in this order:

1. The name of the app. Does it sound interesting or like it does what they are looking for?

2. The icon. You need to have an attractive icon. If your icon sucks, your app probably does too. Or at least that's what people think and then they're gone.

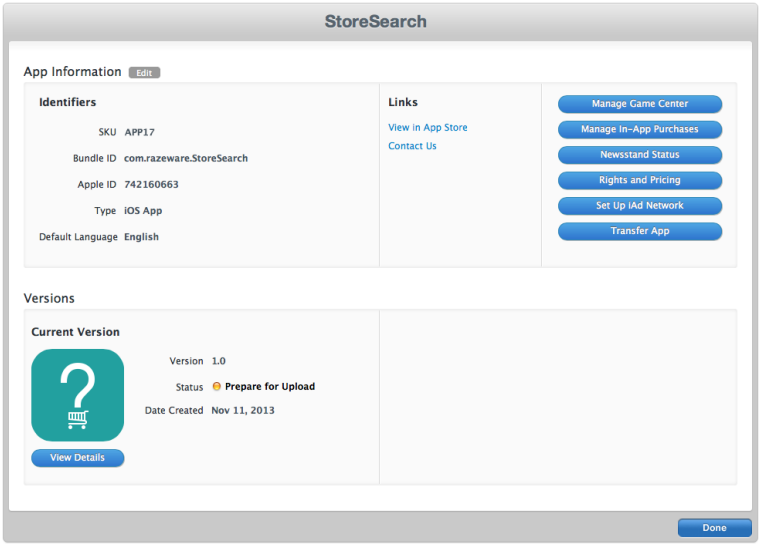
3. The screenshots. You need to have good screenshots that are exciting and that make it clear what your app is about. A lot of developers go further than just regular screenshots; they turn these images into small billboards for their app.

4. If you didn't lose the potential customer in the previous three steps, they might finally read your description for more info.

5. 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, the app will be added to iTunes Connect:



The app is now waiting for upload

The status of the app is "Prepare for Upload". Your app will go through a series of statuses before it becomes available on the store. You can set up iTunes Connect to send you an email each time the status changes, which is something I recommend.

› Click on the **View Details** button to see a summary of the data you have entered. If everything looks good, press the big blue **Ready to Upload Binary** button.

You'll be asked some more questions and then the app's status changes to "Waiting for Upload".

Let's go back to Xcode so we can upload this guy!

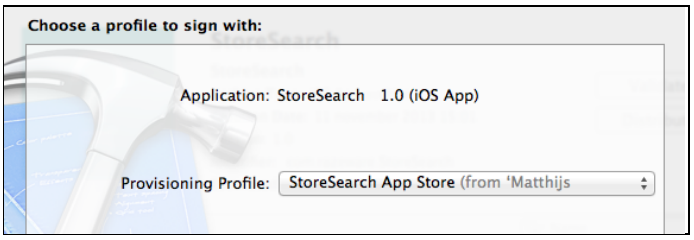
› In the Xcode Organizer, go to the **Archives** tab, select the build you did earlier and choose **Validate**.

There are a bunch of things that can go wrong when you submit your app to the store (for example, forgetting to update your version number when you do an update, or a code signing error) and the Validate option lets you check this from within Xcode, so it's worth doing.

› Enter your Developer Program login credentials.

Xcode will check with iTunes Connect whether there is an application waiting for upload. This is why you had to go to iTunes Connect first.

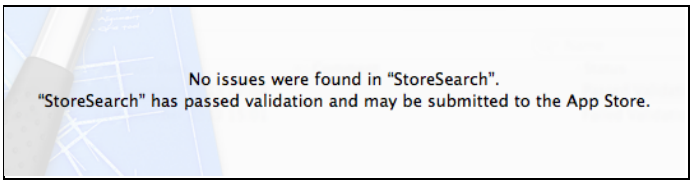
Xcode will now ask you to pick the profile to sign the app with:



The settings for app validation

➤ Choose the **StoreSearch App Store** profile and click the **Validate** button.

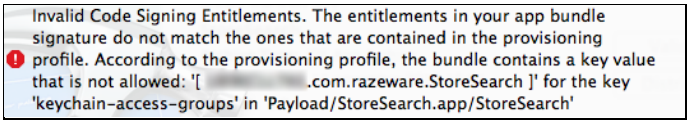
Now wait in agony while the app is being validated…



Yes! This is what we want to see

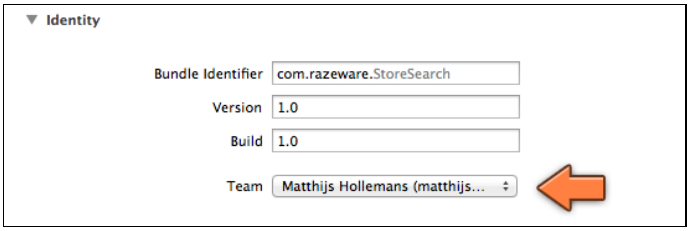
Excellent!

Note: If you're unlucky you might receive an error such as the following:



The app did not validate

If that happens to you, make sure the **Team** is set up properly in the Project Settings screen, under Identity. Then do **Product** → **Archive** again and validate the new archive.

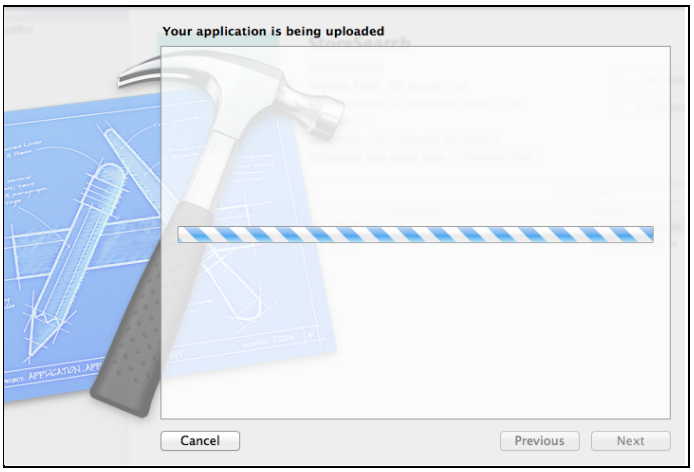


Make sure your team is selected

Now that the app checks out, you can finally submit it. This doesn't guarantee Apple won't reject your app from the store, it just means that it will pass the initial round of validations.

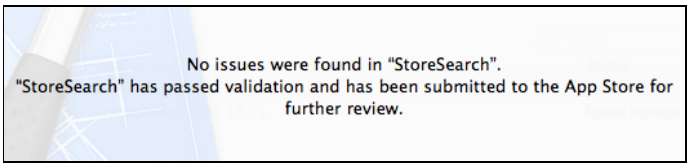
> **Note:** You don't have to submit your source code to Apple, only the final application bundle.

› Select the archive again and click **Distribute**. Choose the **Submit to the iOS App Store** option. Again you're asked for your credentials and the provisioning profile. Fill it out as before and press **Submit** to upload the app.



Exciting!

After a minute or two, you should see a confirmation:



And now the long wait begins...

The status of your app will change to "Upload Received" and it will enter the App Store approval process. If you're lucky the app will go through in a few days, if you're unlucky it can take several weeks. Usually it's about a week or so.

If you find a major bug in the mean time you can reject the file you uploaded (Reject Binary on iTunes Connect) and upload a new one, but this will put you back at square one and you'll have to wait a week again.

If after your app gets approved you want to upload a new version of your app, the steps are largely the same. You go to iTunes Connect and create a new version for the app, fill in some questions, and upload the new binary from Xcode. Updates take about the same amount of time to get reviewed as new apps, so you'll always have to be patient for a few days. (Don't forget to update the version number!)

The end

Awesome, you've done it! You made it all the way through *The iOS Apprentice*. It's been a long journey but I hope you have learned a lot about iPhone and iPad programming and software development in general. I had a lot of fun writing these tutorials and I hope you had a lot of fun reading them!

Because these tutorials are packed with tips and information you may want to go through them again in a few weeks, just to make sure you've picked up on everything!

The world of mobile apps now lies at your fingertips. There is a lot more to be learned about iOS and I encourage you to read the official documentation – it's pretty easy to follow once you understand the basics – and to play around with the myriad of APIs that the iOS SDK has to offer. Most importantly, go write some apps of your own!

Some suggestions for you to start with:

- The iOS Developer Library. You can find this on the iOS Dev Center.
- iOS Technology Overview https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html
- Mobile Human Interface Guidelines (the "HIG"): https://developer.apple.com/library/ios/documentation/userexperience/conceptual/mobilehig/index.html
- iOS App Programming Guide https://developer.apple.com/library/ios/documentation/iphone/conceptual/iphoneosprogrammingguide/Introduction/Introduction.html
- View Controller Programming Guide for iOS http://developer.apple.com/library/ios/#featuredarticles/ViewControllerPGforiPhoneOS/Introduction/Introduction.html
- The WWDC videos. WWDC is Apple's yearly developer conference and the videos of the presentations can be watched online at https://developer.apple.com/videos/. It's really worth it!

If you get stuck, ask for help. Sites such as Stack Overflow (http://stackoverflow.com) and iPhoneDevSDK (http://www.iphonedevsdk.com/forum/) are great, and let's not forget our own Ray Wenderlich forums (http://www.raywenderlich.com/forums).

I often go on Stack Overflow to figure out how to write some code. I usually more-or-less know what I need to do – for example, resize a `UIImage` – and I could spend a few hours figuring out how to do it on my own, but chances are someone else already wrote a blog post about it. Stack Overflow has tons of great tips on almost anything you can do with iOS development.

However, don't post questions like this:

> "i am having very small problem i just want to hide load more data option in tableview after finished loading problem is i am having 23 object in json and i am parsing 5 obj on each time at the end i just want to display three object without load more option."

This is an actual question that I copy-pasted from a forum. That guy isn't going to get any help because a) his question is unreadable; b) he isn't really making it easy for others to help him. Here are some pointers on how to ask effective questions:

- Getting Answers http://www.mikeash.com/getting_answers.html
- What Have You Tried? http://mattgemmell.com/2008/12/08/what-have-you-tried/
- How to Ask Questions the Smart Way http://www.catb.org/~esr/faqs/smart-questions.html

Above all, *have fun programming*, and let me know about your creations!

Credits: This tutorial uses AFNetworking, which was created by Mattt Thompson and Scott Raymond (https://github.com/AFNetworking/AFNetworking).

The shopping cart from the app icon is based on a design from the Noun Project (http://thenounproject.com/).