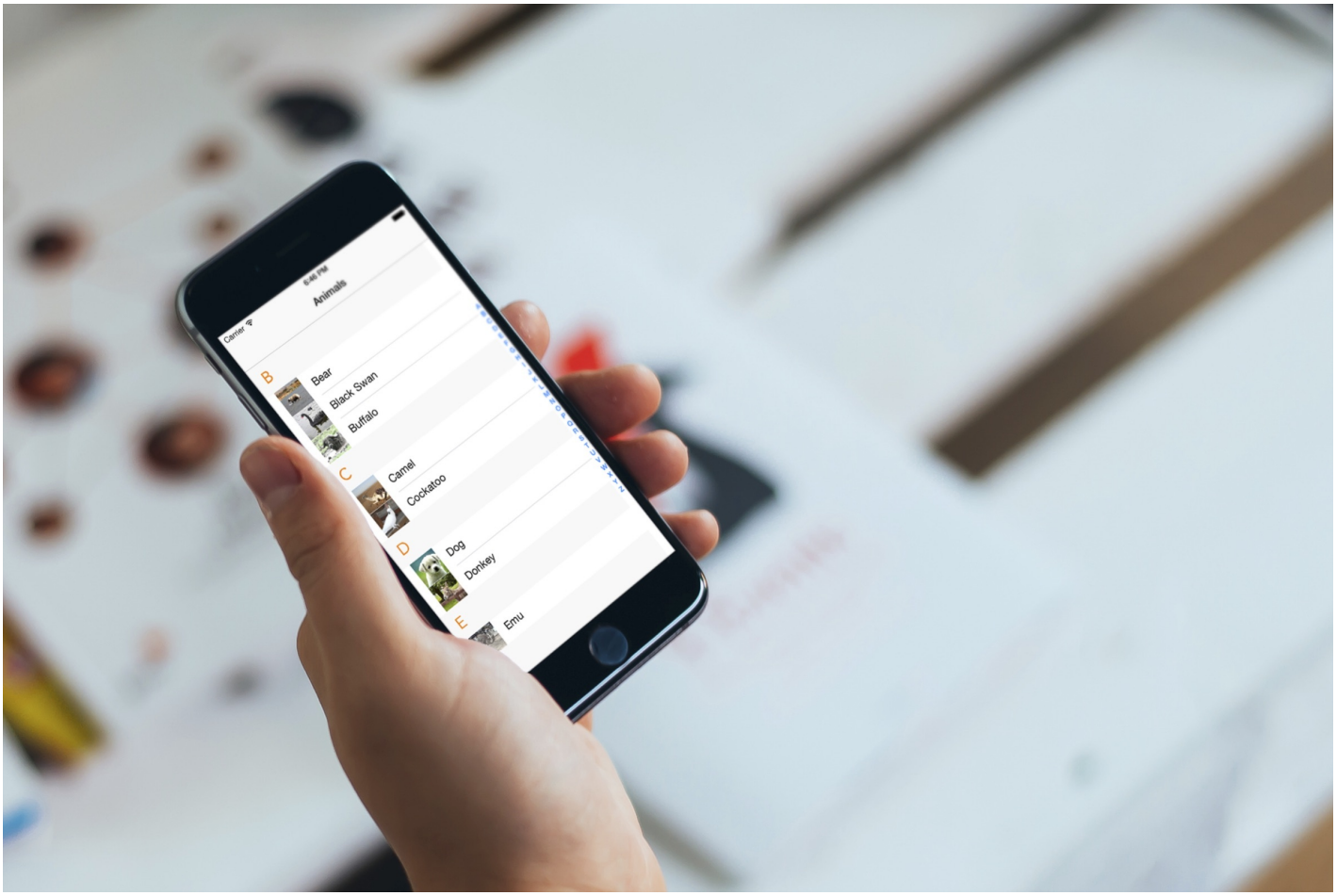
Chapter 2
Adding Sections and Index list in UITableView



If you'd like to show a large number of records in UITableView, you'd best rethink the approach of how to display your data. As the number of rows grows, the table view becomes unwieldy. One way to improve the user experience is to organize the data into sections. By grouping related data together, you offer a better way for users to access it.

Furthermore, you can implement an index list in the table view. An indexed table view is more or less the same as the plain-styled table view. The only difference is that it includes an index on the right side of the table view. An indexed table is very common in iOS apps. The most

well-known example is the built-in Contacts app on the iPhone. By offering index scrolling, users have the ability to access a particular section of the table instantly without scrolling through each section.

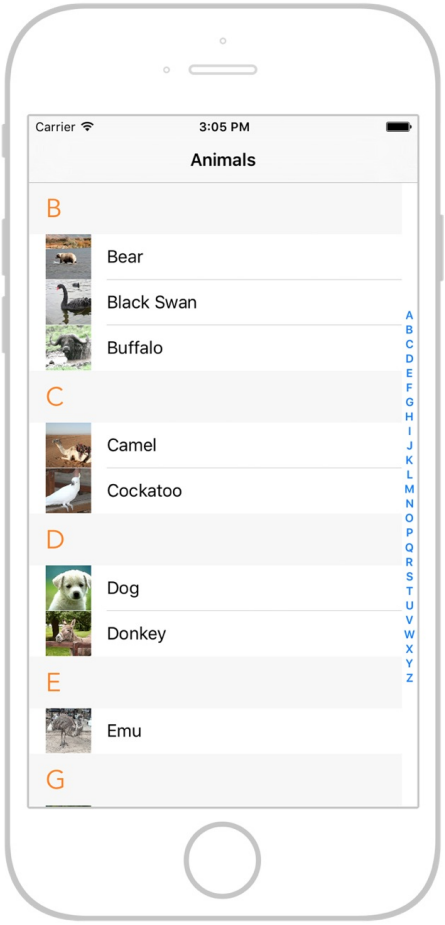
Let's see how we can add sections and an index list to a simple table app. If you have a basic understanding of the `UITableView` implementation, it's not too difficult to add sections and an index list. Basically you need to deal with these methods as defined in the `UITableViewDataSource` protocol:

- *numberOfSectionsInTableView* method – returns the total number of sections in the table view. Usually we set the number of sections to `1`. If you would like to have multiple sections, set this value to a number larger than 1.
- *titleForHeaderInSection* method – returns the header titles for different sections. This method is optional if you do not prefer to assign titles to the section.
- *numberOfRowsInSection* method – returns the total number of rows in a specific section.
- *cellForRowAtIndexPath* method – this method shouldn't be new to you if you know how to display data in UITableView. It returns the table data for a particular section.
- *sectionIndexTitlesForTableView* method – returns the indexed titles that appear in the index list on the right side of the table view. For example, you can return an array of strings containing a value from `A` to `Z`.
- *sectionForSectionIndexTitle* method – returns the section index that the table view should jump to when a user taps a particular index.

There is no better way to explain the implementation than showing you an example. As usual, we will build a simple app, which should give you a better idea of an index list implementation.

A Brief Look at the Demo App

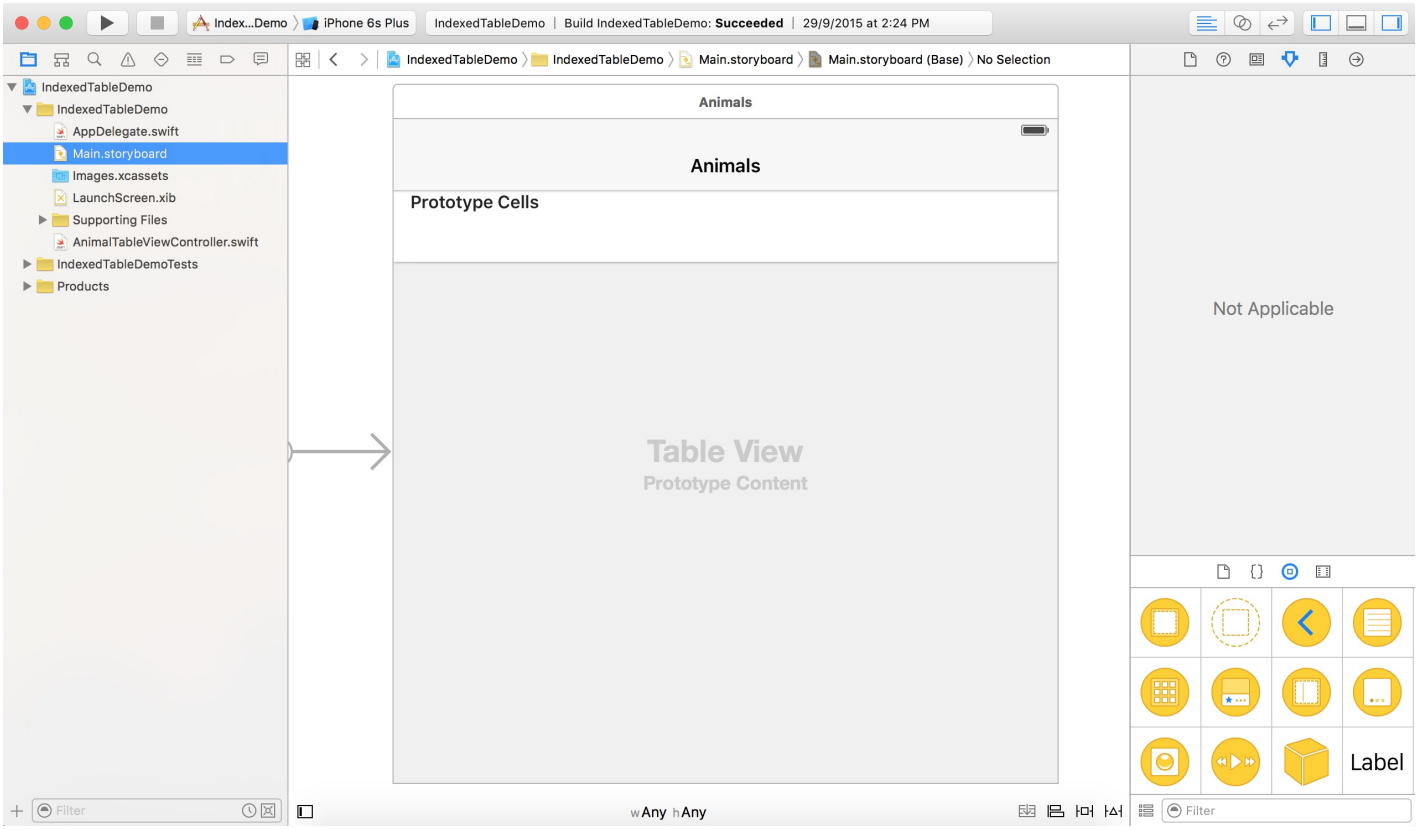
First, let's have a quick look at the demo app that we are going to build. It's a very simple app showing a list of animals in a standard table view. Instead of listing all the animals, the app groups the animals into different sections, and displays an index list for quick access. The screenshot below displays the final deliverable of the demo app.



Download the Xcode Project Template

The focus of this demo is on the implementation of sections and index list. Therefore, instead of building the Xcode project from scratch, you can download the project template from https://www.dropbox.com/s/2whtwzodzw2xrq3/IndexedTableDemoTemplate.zip?dl=0 to start with.

The template already includes everything you need to start with. If you build the template, you'll have an app showing a list of animals in a table view (but without sections and index). Later, we will modify the app, group the data into sections, and add an index list to the table.



Displaying Sections in UITableView

Okay, let's get started. If you open the `IndexTableDemo` project, the animal data is defined in an array:

```
let animals = ["Bear", "Black Swan", "Buffalo", "Camel", "Cockatoo", "Dog",
"Donkey", "Emu", "Giraffe", "Greater Rhea", "Hippopotamus", "Horse", "Koala",
"Lion", "Llama", "Manatus", "Meerkat", "Panda", "Peacock", "Pig", "Platypus",
"Polar Bear", "Rhinoceros", "Seagull", "Tasmania Devil", "Whale", "Whale
Shark", "Wombat"]
```

Well, we're going to organize the data into sections based on the first letter of the animal name. There are a lot of ways to do that. One way is to manually replace the animals array with a dictionary like I've shown below:

```
let animals: [String: [String]] = ["B" : ["Bear", "Black Swan", "Buffalo"],
        "C" : ["Camel", "Cockatoo"],
        "D" : ["Dog", "Donkey"],
        "E" : ["Emu"],
        "G" : ["Giraffe", "Greater Rhea"],
        "H" : ["Hippopotamus", "Horse"],
```

```
        "K" : ["Koala"],
        "L" : ["Lion", "Llama"],
        "M" : ["Manatus", "Meerkat"],
        "P" : ["Panda", "Peacock", "Pig", "Platypus", "Polar Bear"],
        "R" : ["Rhinoceros"],
        "S" : ["Seagull"],
        "T" : ["Tasmania Devil"],
        "W" : ["Whale", "Whale Shark", "Wombat"]]
```

In the above code, we've turned the animals array into a dictionary. The first letter of the animal name is used as a key. The value that is associated with the corresponding key is an array of animal names.

We could manually create the dictionary, but wouldn't it be great if we could create the indexes from the `animals` array? Let's see how it can be done. First, declare two instance variables in the `AnimalTableViewController` class:

```
var animalsDict = [String: [String]]()
var animalSectionTitles = [String]()
```

We initialize an empty dictionary for storing the animals and an empty array for storing the section titles of the table. The section title is the first letter of the animal name (e.g. B).

Because we want to generate a dictionary from the `animals` array, we need a helper method to handle the generation. Insert the following method in the `AnimalTableViewController` class:

```
func createAnimalDict() {
    for animal in animals {
        // Get the first letter of the animal name and build the dictionary
        let animalKey =
animal.substringToIndex(animal.startIndex.advancedBy(1))
        if var animalValues = animalsDict[animalKey] {
            animalValues.append(animal)
            animalsDict[animalKey] = animalValues
        } else {
            animalsDict[animalKey] = [animal]
        }
    }

    // Get the section titles from the dictionary's keys and sort them in
ascending order
    animalSectionTitles = [String](animalsDict.keys)
    animalSectionTitles = animalSectionTitles.sort({ $0 < $1 })
}
```

In this method, we loop through all the items in the `animals` array. For each item, we initially extract the first letter of the animal's name. In Swift, the `subStringToIndex` method of a string can return a new string containing the characters up to a given index. The index should be of the type `String.Index`. To obtain an index for a specific position, you have to ask the string itself for the `startIndex` and then use the global `advance()` function to iterate over all characters between the beginning of the string and the target position. In this case, the target position is `1`, as we are only interested in the first character.

As mentioned before, the first letter of the animal's name is used as a key of the dictionary. The value of the dictionary is an array of animals of that particular key. So once we got the key, we either create a new array of animals or append the item to an existing array. Here we show the values of `animalsDict` for the first four iterations:

- Iteration #1: `animalsDict["B"] = ["Bear"]`
- Iteration #2: `animalsDict["B"] = ["Bear", "Black Swan"]`
- Iteration #3: `animalsDict["B"] = ["Bear", "Black Swan", "Buffalo"]`
- Iteration #4: `animalsDict["C"] = ["Camel"]`

After `animalsDict` is completely generated, we can retrieve the section titles from the keys of the dictionary.

To retrieve the keys of a dictionary, you can simply call the `keys` method. However, the keys returned are unordered. Swift's standard library provides a function called `sort`, which returns a sorted array of values of a known type, based on the output of a sorting closure you provide.

The closure takes two arguments of the same type (in this example, it's the string) and returns a `Bool` value to state whether the first value should appear before or after the second value once the values are sorted. If the first value should appear before the second value, it should return true.

One way to write the sort closure is like this:

```
animalSectionTitles = animalSectionTitles.sort( { (s1:String, s2:String) ->
Bool in
            return s1 < s2
```

```
        })
```

You should be very familiar with the closure expression syntax. In the body of the closure, we compare the two string values. It returns `true` if the second value is greater than the first value. For instance, the value of `s1` is `B` and that of `s2` is `E`. Because B is smaller than E, the closure returns true, indicating that B should appear before E. In this case, we can sort the values in alphabetical order.

If you read the earlier code snippet carefully, you may wonder why I wrote the `sort` closure like this:

```
animalSectionTitles = animalSectionTitles.sort({ $0 < $1 })
```

It's a shorthand in Swift for writing inline closures. Here `$0` and `$1` refer to the first and second String arguments. If you use shorthand argument names, you can omit nearly everything of the closure including argument list and in keyword; you will just need to write the body of the closure.

In Swift 2, Apple introduces another sort function called `sortInPlace`. This new function is very similar to the `sort` function. Instead of returning you a sorted array, the `sortInPlace` function applies the sorting on the original array. You can replace the line of code with the one below:

```
animalSectionTitles.sortInPlace({ $0 < $1 })
```

With the helper method created, update the `viewDidLoad` method to call it up:

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

    // Generate the animal dictionary
    createAnimalDict()
}
```

Next, change the `numberOfSectionsInTableView` method and return the total number of sections:

```
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    // Return the number of sections.
```

```
    return animalSectionTitles.count
}
```

To display a header title in each section, we need to implement the `titleForHeaderInSection` method. This method is called every time a new section is displayed. Based on the given section index, we simply return the corresponding section title.

```
override func tableView(tableView: UITableView, titleForHeaderInSection
section: Int) -> String? {
    return animalSectionTitles[section]
}
```

It's very straightforward, right? Next, we have to tell the table view the number of rows in a particular section. Update the `numberOfRowsInSection` method in `AnimalTableViewController.swift` like this:

```
override func tableView(tableView: UITableView, numberOfRowsInSection section:
Int) -> Int {
    // Return the number of rows in the section.
    let animalKey = animalSectionTitles[section]
    if let animalValues = animalsDict[animalKey] {
        return animalValues.count
    }

    return 0
}
```

When the app starts to render the data in the table view, the `numberOfRowsInSection` method is called every time a new section is displayed. Based on the section index, we can get the section title and use it as a key to retrieve the animal names of that section, followed by returning the total number of animal names for that section.

Lastly, modify the `cellForRowAtIndexPath` method as follows:

```
override func tableView(tableView: UITableView, cellForRowAtIndexPath
indexPath: NSIndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("Cell",
forIndexPath: indexPath)

    // Configure the cell...
    let animalKey = animalSectionTitles[indexPath.section]
    if let animalValues = animalsDict[animalKey] {
        cell.textLabel?.text = animalValues[indexPath.row]
```

```
        // Convert the animal name to lower case and
        // then replace all occurrences of a space with an underscore
        let imageFilename =
animalValues[indexPath.row].lowercaseString.stringByReplacingOccurrencesOfString
 ", withString: "_", options: [], range: nil)
        cell.imageView?.image = UIImage(named: imageFilename)
    }


    return cell
}
```

The `indexPath` argument contains the current row number, as well as, the current section index. So, based on the section index, we retrieve the section title (e.g. "B") and use it as the key to retrieve the animal names for that section. The rest of the code is very straightforward. We simply get the animal name and set it as the cell label. The `imageFilename` variable is computed by converting the animal name to lowercase letters, followed by replacing all occurrences of a space with an underscore.

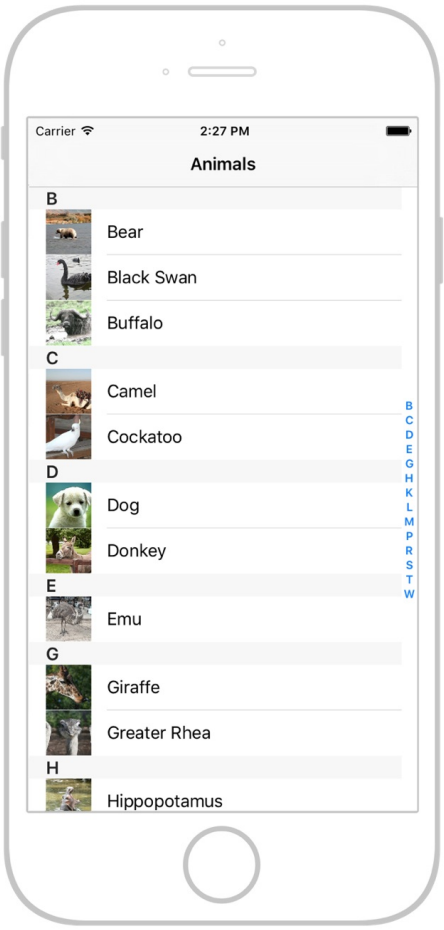
Okay, you're ready to go! Hit the Run button and you should end up with an app with sections but without the index list.

Adding An Index List to UITableView

Cool, right? But how can you add an index list to the table view? Again it's easier than you thought and can be achieved with just a few lines of code. Simply add the `sectionIndexTitlesForTableView` method and return an array of section indexes. Here we will use the section titles as the indexes.

```
override func sectionIndexTitlesForTableView(tableView: UITableView) ->
[String]? {
    return animalSectionTitles
}
```

That's it! Compile and run the app again. You should find the index on the right side of the table. Interestingly, you do not need any implementation and the indexing already works! Try to tap any of the indexes and you'll be brought to a particular section of the table.



Adding An A-Z Index List

Looks like we've done everything. So why did we mention the `sectionForSectionIndexTitle` method at the very beginning?

Currently, the index list doesn't contain the entire alphabet. It just shows those letters that are defined as the keys of the `animals` dictionary. Sometimes, you may want to display A-Z in the index list. Let's declare a new variable named `animalIndexTitles` in `AnimalTableViewController.swift`:

```
let animalIndexTitles = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K",
"L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z"]
```

Next, change the `sectionIndexTitlesForTableView` method and return the `animalIndexTitles` array instead of the `animalSectionTitles` array.

```
override func sectionIndexTitlesForTableView(tableView: UITableView) ->
```

```
[AnyObject]! {
    return animalIndexTitles
}
```

Now, compile and run the app again. Cool! The app displays the index from A to Z.

But wait a minute... It doesn't work properly! If you try tapping the index "C," the app jumps to the "D" section. And if you tap the index "G," it directs you to the "K" section. Below shows the mapping between the old and new indexes.

Well, as you may notice, the number of indexes is greater than the number of sections, and the `UITableView` object doesn't know how to handle the indexing. It's your responsibility to implement the `sectionForSectionIndexTitle` method and explicitly tell the table view the section number when a particular index is tapped. Add the following new method:

```
override func tableView(tableView: UITableView, sectionForSectionIndexTitle
title: String, atIndex index: Int) -> Int {

    guard let index = animalSectionTitles.indexOf(title) else {
        return -1
    }

    return index
}
```

Based on the selected index name (i.e. title), we locate the correct section index of `animalSectionTitles`. In Swift, you use the method called `indexOf` to find the index of a particular item in the array. If it's not found, we return `-1`. Otherwise, we return the index of the item. Compile and run the app again. The index list should now work!

```
Note: The old find function is not supported any more with Swift 2.0!
```

Customizing Section Headers

You can easily customize the section headers by overriding some of the methods defined in the

`UITableView` class and the `UITableViewDelegate` protocol. In this demo, we'll make two simple changes:

- Alter the height of the section header
- Change the font of the section header

To alter the height of the section header, you can simply override the `heightForHeaderInSection` method and return the preferred height:

```
override func tableView(tableView: UITableView, heightForHeaderInSection
section: Int) -> CGFloat {
    return 50
}
```

Before the section header view is displayed, the `willDisplayHeaderView` method will be called. The method includes an argument named view. This view object can be a custom header view or a standard one. In our demo, we just use the standard header view, which is the `UITableViewHeaderFooterView` object. Once you have the header view, you can alter the text color and font accordingly.

```
override func tableView(tableView: UITableView, willDisplayHeaderView view:
UIView, forSection section: Int) {
    let headerView = view as! UITableViewHeaderFooterView
    headerView.textLabel?.textColor = UIColor.orangeColor()
    headerView.textLabel?.font = UIFont(name: "Avenir", size: 25.0)
}
```

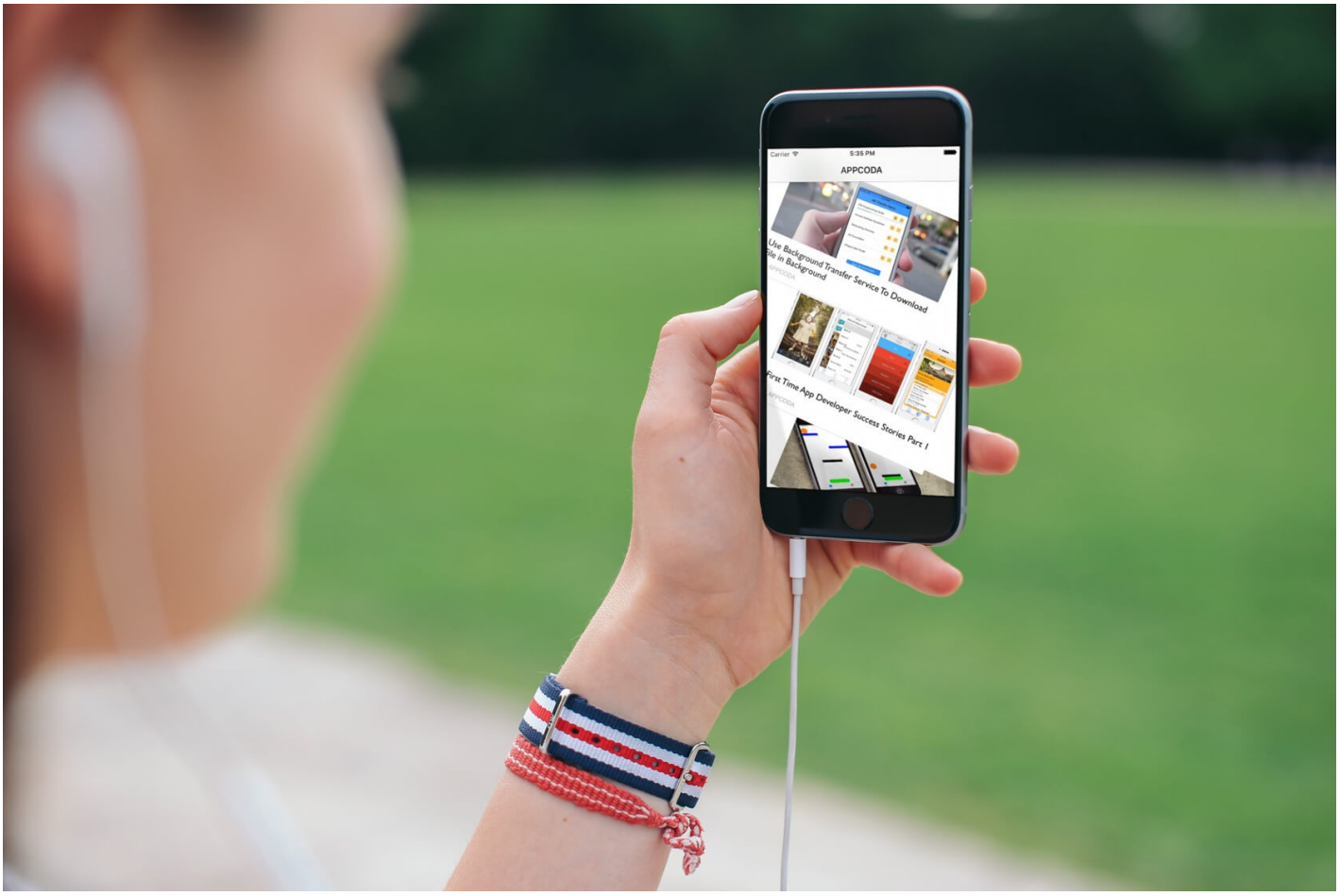
Run the app again. The header view should be updated with your preferred font and color.

Summary

When you need to display a large number of records, it is simple and effective to organize the data into sections and provide an index list for easy access. In this chapter, we've walked you through the implementation of an indexed table. By now, I believe you should know how to add sections and an index list to your table view.

For your reference, you can download the complete Xcode project from https://www.dropbox.com/s/kpd0q1m5ccsaup7/IndexedTableDemo.zip?dl=0.

Chapter 3
Animating Table View Cells



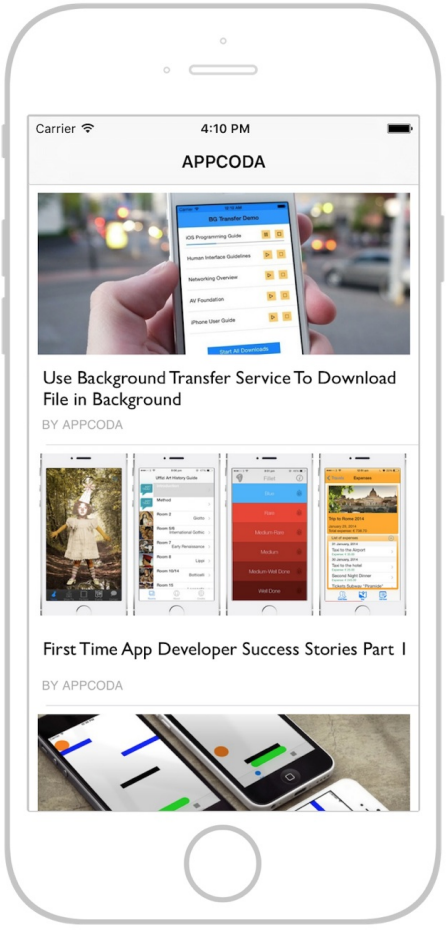
When you read this chapter, I assume you already knew how to use `UITableView` to present data. If not, go back and read the Beginning iOS 9 Programming with Swift book.

The `UITableView` class provides a powerful way to present information in table form, and it is one of the most commonly used components in iOS apps. Whether you're building a productivity app, to-do app, or social app, you would make use of table views in one form or another. The default implementation of `UITableView` is preliminary and only suitable for basic apps. To differentiate one's app from the rest, you usually provide customizations for the table views and table cells in order to make the app stand out. In this chapter, we'll show you a powerful technique to liven up your app by adding subtle animation.

The Google+ app is a great example of table view animation. If you've used the app, every table row (or card) is animated as you scroll through the table. The card seems to slide in from the side when it first appears. This subtle animation would greatly enhance the user experience of your app.

It is very easy to animate a table view cell. Again, to demonstrate how the animation is done, we'll tweak an existing table-based app and add a subtle animation.

To start with, first download the project template from https://www.dropbox.com/s/qnxqfwppra62mg8/TableCellAnimationTemplate.zip?dl=0. After downloading, compile the app and make sure you can run it properly. It's just a very simple app displaying a list of articles.



Creating a Simple Fade-in Animation for Table View Cells

Let's start by tweaking the table-based app with a simple fade-in effect. So how can we add this

subtle animation when the table row appears? If you look into the documentation of the `UITableViewDelegate` protocol, you should find a method called `tableView(_:willDisplayCell:forRowAtIndexPath:):`

```
optional func tableView(_ tableView: UITableView, willDisplayCell cell:
UITableViewCell, forRowAtIndexPath indexPath: NSIndexPath)
```

The method will be called right before a row is drawn. By implementating the method, you can customize the cell object and add your own animation before the cell is displayed. Here is what you need to create the fade-in effect. Insert the code snippet in `ArticleTableViewController.swift`:

```
override func tableView(tableView: UITableView, willDisplayCell cell:
UITableViewCell, forRowAtIndexPath indexPath: NSIndexPath) {

    // Define the initial state (Before the animation)
    cell.alpha = 0

    // Define the final state (After the animation)
    UIView.animateWithDuration(1.0, animations: { cell.alpha = 1 })
}
```

Core Animation provides iOS developers with an easy way to create animation. All you need to do is define the initial and final state of the visual element. Core Animation will then figure out the required animation between these two states.

In the above code, we first set the initial alpha value of the cell to `0`, which represents total transparency. Then we begin the animation; set the duration to 1 second and define the final state of the cell, which is completely opaque. This will automatically create a fade-in effect when the table cell appears.

You can now compile and run the app. Scroll through the table view and enjoy the fade-in animation.

Creating a Rotation Effect Using CATransform3D

Easy, right? With a few lines of code, your app looks a bit different than a standard table-based app. The `tableView(_:willDisplayCell:forRowAtIndexPath:)` method is the key to table view cell animation. You can implement whichever type of animation in the method. The fade-in

animation is very simple. Now let's try to implement another animation using `CATransform3D`. Don't worry, you just need a few lines of code.

To add a rotation effect to the table cell, update the method like this:

```
override func tableView(tableView: UITableView, willDisplayCell cell:
UITableViewCell, forRowAtIndexPath indexPath: NSIndexPath) {

    // Define the initial state (Before the animation)
    let rotationAngleInRadians = 90.0 * CGFloat(M_PI/180.0)
    let rotationTransform = CATransform3DMakeRotation(rotationAngleInRadians,
0, 0, 1)
    cell.layer.transform = rotationTransform

    // Define the final state (After the animation)
    UIView.animateWithDuration(1.0, animations: { cell.layer.transform =
CATransform3DIdentity })
}
```

Same as before, we define the initial and final state of the transformation. The general idea is that we first rotate the cell by 90 degrees clockwise and then bring it back to the normal orientation which is the final state.

Okay, but how can we rotate a table cell by 90 degrees clockwise?

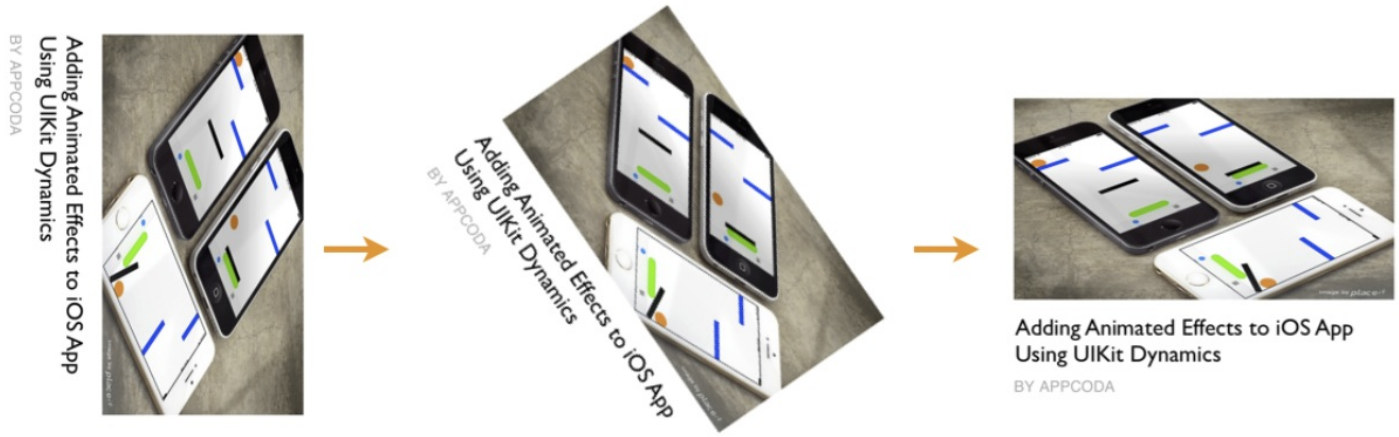
The key is to use the `CATransform3DMakeRotation` function to create the rotation transform. The function takes four parameters:

- Angle in radians - this is the angle of rotation. As the angle is in radian, we first need to convert the degrees into radians.
- X axis - this is the axis that goes from the left of the screen to the right of the screen.
- Y axis - this is the axis that goes from the top of the screen to the bottom of the screen.
- Z axis - this is the axis that points directly out of the screen.

Since the rotation is around the Z axis, we set the value of this parameter to `1`, while leaving the value of the X axis and Y axis at `0`. Once we create the transform, it is assigned to the cell's layer.

Next, we start the animation with the duration of 1 second. The final state of the cell is set to `CATransform3DIdentity`, which will reset the cell to the original position.

Okay, hit Run to test the app!



Quick Tip: You may wonder what CATransform3D is. It is actually a structure representing a matrix. Performing transformation in 3D space such as rotation, involves some matrices calculation. I'll not go into the details of matrices calculation. If you want to learn more, you can check out http://www.matrix44.net/cms/notes/opengl-3d-graphics/basic-3d-math-matrices.

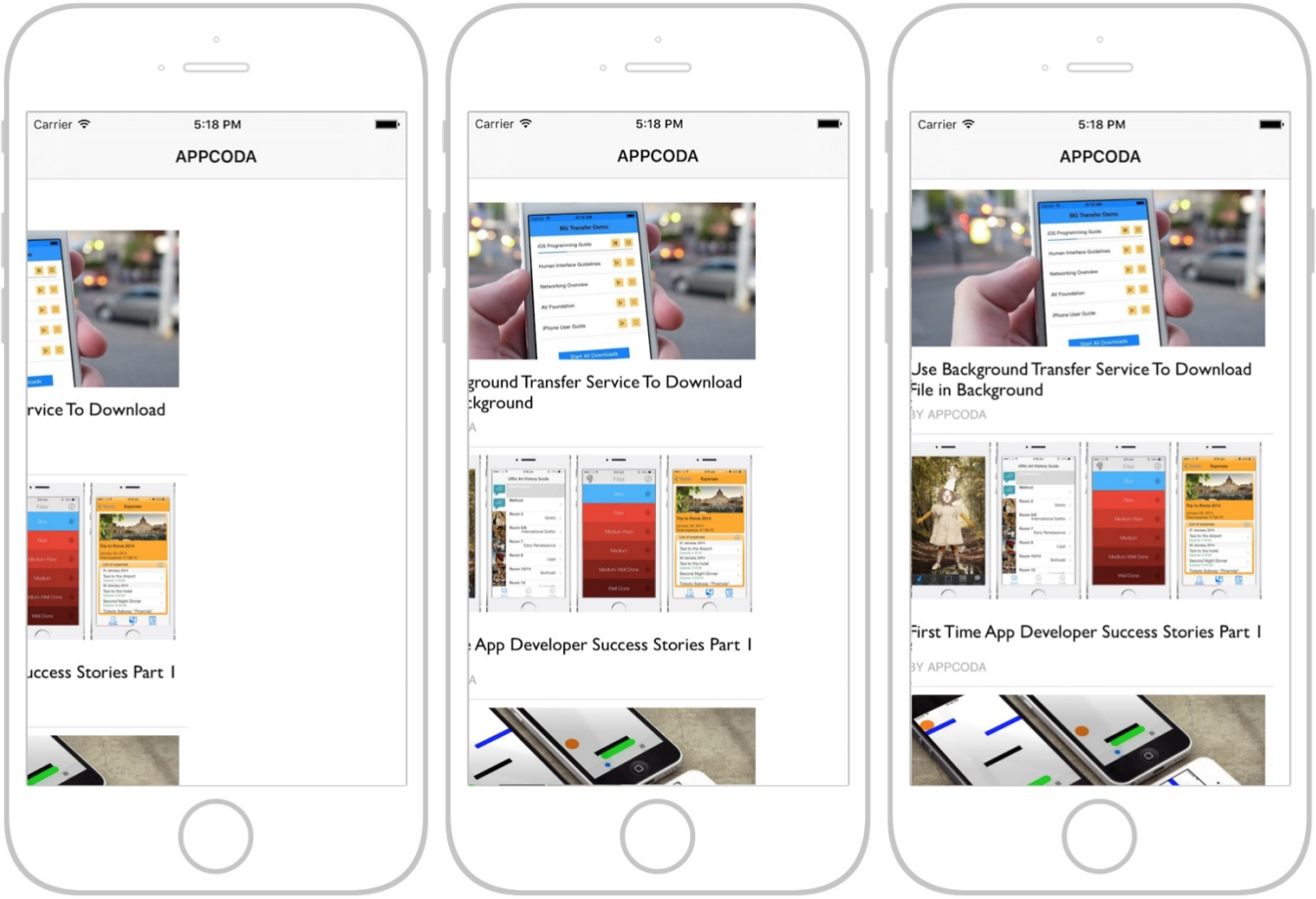
Creating a Fly-in Effect using CATransform3DTranslate

Does the rotation effect look cool? You can further tweak the animation to make it even better. Try to change the `tableView(_:willDisplayCell:forRowAtIndexPath:)` method and replace the initialization of `rotationTransform` with the following line of code:

```
let rotationTransform = CATransform3DTranslate(CATransform3DIdentity, -500,
100, 0)
```

The line of code simply translates or shifts the position of the cell. It indicates the cell is shifted to the left (negative value) by `500` points and down (positive value) by `100` points. There is no change in the Z axis.

Now you're ready to test the app again. Hit the Run button and play around with the fly-in effect.



Your Exercise

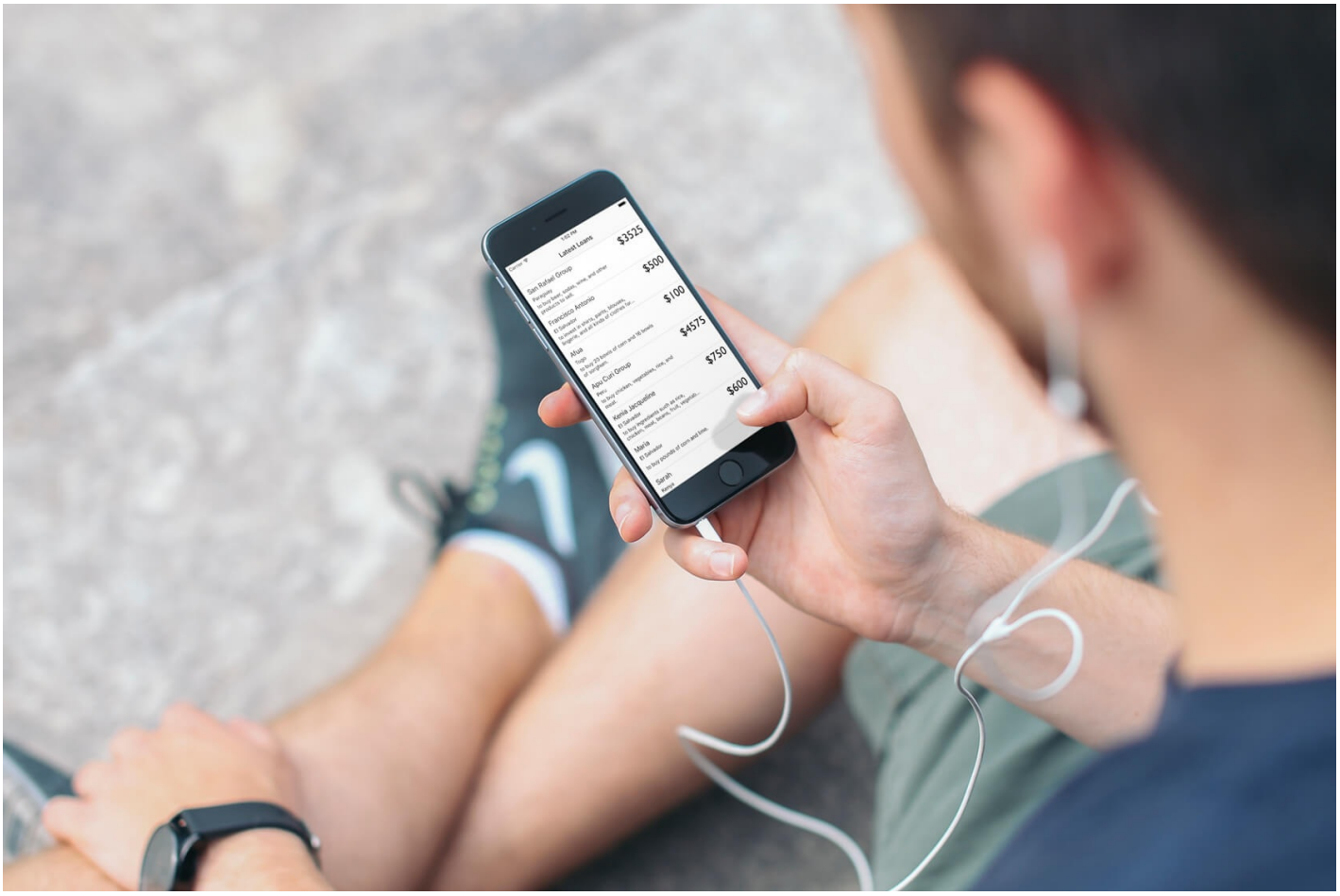
For now, the cell animation is shown every time you scroll through the table, whether you're scrolling down or up the table view. Though the animation is nice, your user will find it annoying if the animation is displayed too frequently. You may want to display the animation only when the cell first appears. Try to modify the existing project and add that restriction.

Summary

In this chapter, I just showed you the basics of table cell animation. Try to change the values of the transform and see what effects you get.

For reference, you can download the complete Xcode project from https://www.dropbox.com/s/98s9wiwe38l81sc/TableCellAnimation.zip?dl=0. The solution of the exercise is included in the project.

Chapter 4
Working with JSON



First, what's JSON? JSON (short for JavaScript Object Notation) is a text-based, lightweight, and easy way for storing and exchanging data. It's commonly used for representing structural data and data interchange in client-server applications, serving as an alternative to XML. A lot of the web services we use everyday have JSON-based APIs. Most of the iOS apps, including Twitter, Facebook, and Flickr send data to their backend web services in JSON format. As an example, here is a JSON representation of a sample Movie object:

```
{
    "title": "The Amazing Spider-man",
    "release_date": "03/07/2012",
    "director": "Marc Webb",
```

```
    "cast": [
        {
            "name": "Andrew Garfield",
            "character": "Peter Parker"
        },
        {
            "name": "Emma Stone",
            "character": "Gwen Stacy"
        },
        {
            "name": "Rhys Ifans",
            "character": "Dr. Curt Connors"
        }
    ]
}
```

As you can see, JSON formatted data is more human-readable and easier to parse than XML. I'll not go into the details of JSON. This is not the purpose of this chapter. If you want to learn more about the technology, I recommend you tocheck out the JSON Guide at http://www.json.org/.

Since the release of iOS 5, the iOS SDK has already made it easy for developers to fetch and parse JSON data. It comes with a handy class called `NSJSONSerialization`, which can automatically convert JSON formatted data to objects. Later in this chapter, I will show you how to use the API to parse some sample JSON formatted data, returned by a web service. Once you understand how it works, it is fairly easy to build an app by integrating with other free/paid web services.

Demo App

As usual, we'll create a demo app. Let's call it *KivaLoan*. The reason why we name the app *KivaLoan* is that we will utilize a JSON-based API provided by Kiva.org. If you haven't heard of Kiva, it is a non-profit organization with a mission to connect people through lending to alleviate poverty. It lets individuals lend as little as $25 to help create opportunities around the world. Kiva provides free web-based APIs for developers to access their data. For our demo app, we'll call up the following Kiva API to retrieve the most recent fundraising loans and display them in a table view:

```
https://api.kivaws.org/v1/loans/newest.json
```

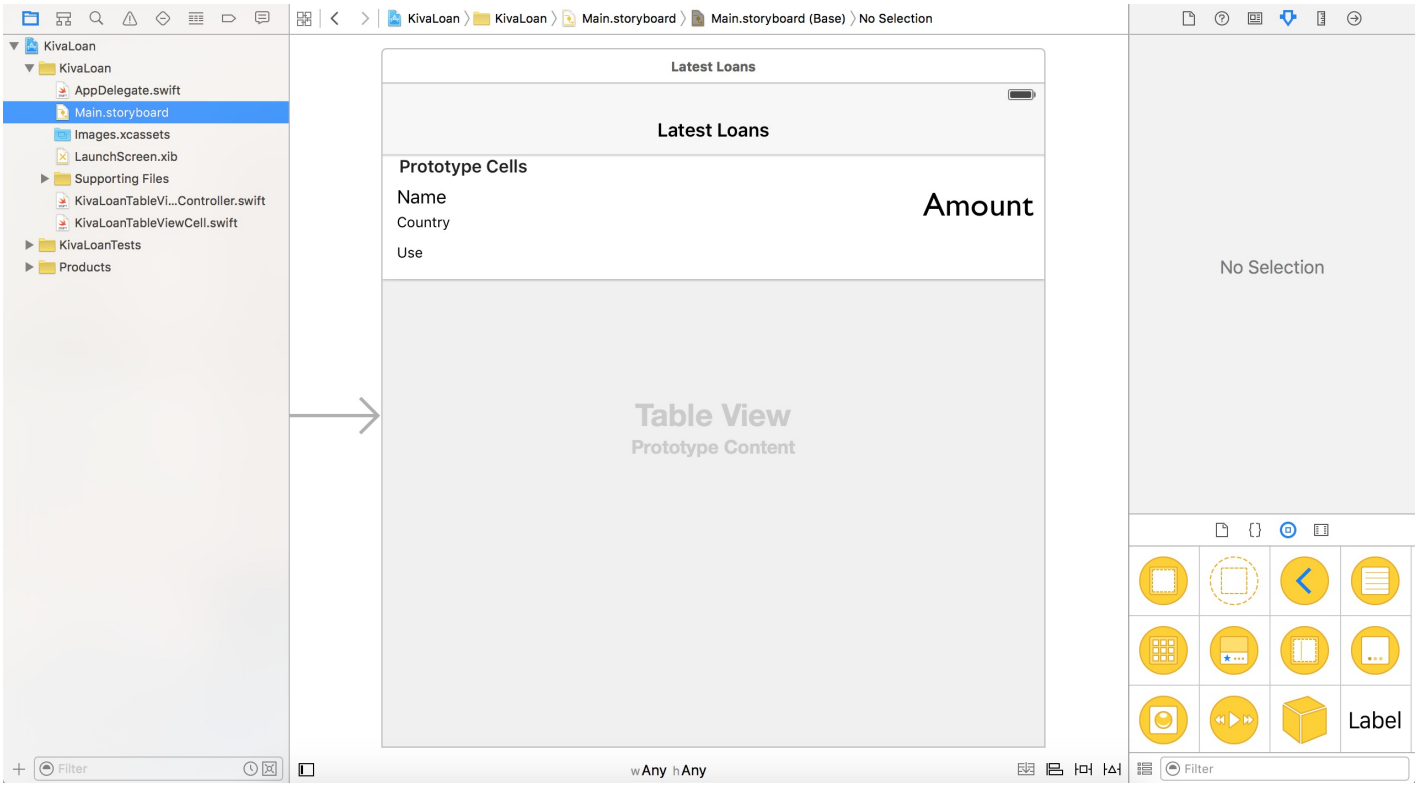
Quick note: Starting from iOS 9, Apple introduced a new feature called App Transport Security with the aim to improve the security of connections between an app and web services. By default, all outgoing connections should ride on HTTPS. Otherwise, your app will not be allowed to connect to the web service.

The returned data of the above API is in JSON format. Here is a sample result:

```
loans: (
        {
        activity = Retail;
        "basket_amount" = 0;
        "bonus_credit_eligibility" = 0;
        "borrower_count" = 1;
        description =         {
            languages =             (
                fr,
                en
            );
        };
        "funded_amount" = 0;
        id = 734117;
        image =         {
            id = 1641389;
            "template_id" = 1;
        };
        "lender_count" = 0;
        "loan_amount" = 750;
        location =         {
            country = Senegal;
            "country_code" = SN;
            geo =             {
                level = country;
                pairs = "14 -14";
                type = point;
            };
        };
        name = "Mar\U00e8me";
        "partner_id" = 108;
        "planned_expiration_date" = "2014-08-05T09:20:02Z";
        "posted_date" = "2014-07-06T09:20:02Z";
        sector = Retail;
        status = fundraising;
        use = "to buy fabric to resell";
    },
....
....
)
```

You will learn how to use the `NSJSONSerialization` class to convert the JSON formatted data into objects. It's unbelievably simple. You'll see what I mean in a while.

To keep you focused on learning the JSON implementation, you can first download the project template from https://www.dropbox.com/s/qud9s0zfcya3ji2/KivaLoanTemplate.zip?dl=0. I have already created the skeleton of the app for you. It is a simple table-based app that displays a list of loans provided by Kiva.org. The project template includes a pre-built storyboard and custom classes for the table view controller and prototype cell. If you run the template, it should result in an empty table app.



Creating JSON Data Model

We will first create a class to model a loan. It's not requried for loading JSON but a best practice to create a separate class (or structure) for storing the data model. The `Loan` class represents the loan information in the KivaLoan app and is used to store the loan information returned by Kiva.org. To keep things simple, we won't use all the returned data of a loan. Instead, the app will just display the following fields of a loan:

- Name of the loan applicant

```
name = "Mar\U00e8me";
```

- Country of the loan applicant

```
location =         {
            country = Senegal;
            "country_code" = SN;
            geo =             {
                level = country;
                pairs = "14 -14";
                type = point;
            };
        };
```

- How the loan will be used

```
use = "to buy fabric to resell";
```

- Amount

```
"loan_amount" = 750;
```

These fields are good enough for filling up the labels in the table view. Now create a new class file using the Swift File template. Name it `Loan.swift` and declare the `Loan` class like this:

```
class Loan {

    var name:String = ""
    var country:String = ""
    var use:String = ""
    var amount:Int = 0

}
```

JSON supports a few basic data types including number, String, Boolean, Array, and Objects (an associated array with key and value pairs).

For the loan fields, the loan amount is stored as a numeric value in the JSON-formatted data. This is why we declared the `amount` property with the type `Int`. For the rest of the fields, they are declared with the type `String`.

Fetching Loans with the Kiva API

As I mentioned earlier, the Kiva API is free to use. No registration is required. You may point your browser to the following URL and you'll get the latest fundraising loans in JSON format.

```
https://api.kivaws.org/v1/loans/newest.json
```

Okay, let's see how we can call up the Kiva API and parse the returned data. First, open `KivaLoanTableViewController.swift` and declare two variables at the very beginning:

```
let kivaLoadURL = "https://api.kivaws.org/v1/loans/newest.json"
var loans = [Loan]()
```

We just defined the URL of the Kiva API, and declare the `loans` variable for storing an array of Loan objects. Next, insert the following methods in the same file:

```
func getLatestLoans() {
    let request = NSURLRequest(URL: NSURL(string: kivaLoadURL)!)
    let urlSession = NSURLSession.sharedSession()
    let task = urlSession.dataTaskWithRequest(request, completionHandler: {
(data, response, error) -> Void in

        if let error = error {
            print(error)
            return
        }

        // Parse JSON data
        if let data = data {
            self.loans = self.parseJsonData(data)

            // Reload table view
            NSOperationQueue.mainQueue().addOperationWithBlock({ () -> Void in
                self.tableView.reloadData()
                })

        }

    })

    task.resume()
}

func parseJsonData(data: NSData) -> [Loan] {
```

```
    var loans = [Loan]()

    do {
        let jsonResult = try NSJSONSerialization.JSONObjectWithData(data,
options: NSJSONReadingOptions.MutableContainers) as? NSDictionary

        // Parse JSON data
        let jsonLoans = jsonResult?["loans"] as! [AnyObject]
        for jsonLoan in jsonLoans {
            let loan = Loan()
            loan.name = jsonLoan["name"] as! String
            loan.amount = jsonLoan["loan_amount"] as! Int
            loan.use = jsonLoan["use"] as! String
            let location = jsonLoan["location"] as! [String:AnyObject]
            loan.country = location["country"] as! String
            loans.append(loan)
        }

    } catch {
        print(error)
    }

    return loans
}
```

These two methods form the core part of the app. Both methods work collaboratively to call the Kiva API, retrieve the latest loans in JSON format and translate the JSON-formatted data into an array of `Loan` objects. Let's go through them in detail.

In the `getLatestLoans` method, we first create an instance of `NSURLSession` with the Kiva API. The `NSURLSession` class was introduced in iOS 7 as a successor to `NSURLConnection`, which has been around for several years. `NSURLSession` provides more features, flexibility and power when dealing with online content over HTTP. One improvement of `NSURLSession` is session tasks, which handle the loading of data, as well as uploading and downloading files and data fetching from servers (e.g. JSON data fetching). With sessions, you can schedule three types of tasks: data tasks (`NSURLSessionDataTask`) for retrieving data to memory, download tasks (`NSURLSessionDownloadTask`) for downloading a file to disk, and upload tasks (`NSURLSessionUploadTask`) for uploading a file from disk. Here we use the data task to retrieve contents from Kiva.org. To add a data task to the session, we call the `dataTaskWithURL` method with the specified URL of Kiva. To initiate the data task, you call the resume method (i.e. `task.resume()`). Like most networking APIs, the `NSURLSession` API is asynchronous. Once the

request completes, it returns the data by calling your completion handler closure.

In the completion handler, immediately after the data is returned, we check for an error and invoke the `parseJsonData` method. The data returned is in JSON format. We create a helper method called `parseJsonData` for converting the given JSON-formatted data into an array of `Loan` objects. The Foundation framework provides the `NSJSONSerialization` class, which is capable of converting JSON to Foundation objects and converting Foundation objects to JSON. In the code snippet, we call the `JSONObjectWithData` method with the given JSON data to perform the conversion.

When converting JSON formatted data to objects, the top-level item is usually converted to a Dictionary or an Array. In this case, the top level of the returned data of the Kiva API is converted to a dictionary. You can access the array of loans using the key `loans`. How do you know what key to use? You can either refer to the API documentation or test the JSON data using a JSON browser (e.g. http://jsonviewer.stack.hu). If you've loaded the Kiva API into the JSON browser, here is an excerpt of the result:

```
{
  "paging": {
    "page": 1,
    "total": 5297,
    "page_size": 20,
    "pages": 265
  },
  "loans": [
    {
      "id": 794429,
      "name": "Joel",
      "description": {
        "languages": [
          "es",
          "en"
        ]
      },
      "status": "fundraising",
      "funded_amount": 0,
      "basket_amount": 0,
      "image": {
        "id": 1729143,
        "template_id": 1
      },
      "activity": "Home Appliances",
      "sector": "Personal Use",
```

```
    "use": "To buy home appliances.",
    "location": {
      "country_code": "PE",
      "country": "Peru",
      "town": "Ica",
      "geo": {
        "level": "country",
        "pairs": "-10 -76",
        "type": "point"
      }
    },
    "partner_id": 139,
    "posted_date": "2014-11-20T08:50:02Z",
    "planned_expiration_date": "2015-01-04T08:50:02Z",
    "loan_amount": 400,
    "borrower_count": 1,
    "lender_count": 0,
    "bonus_credit_eligibility": true,
    "tags": [

    ]
  },
  {
    "id": 797222,
    "name": "Lucy",
    "description": {
      "languages": [
        "en"
      ]
    },
    "status": "fundraising",
    "funded_amount": 0,
    "basket_amount": 0,
    "image": {
      "id": 1732818,
      "template_id": 1
    },
    "activity": "Farm Supplies",
    "sector": "Agriculture",
    "use": "To purchase a biogas system for clean cooking",
    "location": {
      "country_code": "KE",
      "country": "Kenya",
      "town": "Gatitu",
      "geo": {
        "level": "country",
        "pairs": "1 38",
        "type": "point"
      }
```

```
      },
      "partner_id": 436,
      "posted_date": "2014-11-20T08:50:02Z",
      "planned_expiration_date": "2015-01-04T08:50:02Z",
      "loan_amount": 800,
      "borrower_count": 1,
      "lender_count": 0,
      "bonus_credit_eligibility": false,
      "tags": [

      ]
    },

     ...
```

As you can see from the above code, `paging` and `loans` are two of the top-level items. Once the `NSJSONSerialization` class converts the JSON data, the result (i.e. `jsonResult`) is returned as a Dictionary with the top-level items as keys. This is why we can use the key `loans` to access the array of loans. Here is the line of code for your reference:

```
let jsonLoans = jsonResult?["loans"] as! [AnyObject]
```

With the array of loans (i.e. jsonLoans) returned, we loop through the array. Each of the array items (i.e. jsonLoan) is converted into a dictionary. In the loop, we extract the loan data from each of the dictionaries and save them in a `Loan` object. Again, you can find the keys (highlighted in yellow) by studying the JSON result. The value of a particular result is stored as `AnyObject`. `AnyObject` is used because a JSON value could be a String, Double, Boolean, Array, Dictionary or null. This is why you have to downcast the value to a specific type such as `String` and `Int`. Lastly, we put the `loan` object into the `loans` array, which is the return value of the method.

```
for jsonLoan in jsonLoans {
    let loan = Loan()
    loan.name = jsonLoan["name"] as! String
    loan.amount = jsonLoan["loan_amount"] as! Int
    loan.use = jsonLoan["use"] as! String
    let location = jsonLoan["location"] as! [String:AnyObject]
    loan.country = location["country"] as! String
    loans.append(loan)
}
```

After the JSON data is parsed and the array of loans is returned, we call the `reloadData`

method to reload the table. You may wonder why we need to call `NSOperationQueue.mainQueue().addOperationWithBlock` and execute the data reload in the main thread. The block of code in the completion handler of the data task is executed in a background thread. If you just call the `reloadData` method in a background thread, the data reload will not happen immediately. To ensure a responsive GUI update, this operation should be performed in the main thread. This is why we call the `NSOperationQueue.mainQueue().addOperationWithBlock` method and request to run the `reloadData` method in the main queue.

```
NSOperationQueue.mainQueue().addOperationWithBlock({ () -> Void in
    self.tableView.reloadData()
    })
```

Quick note: You can also use dispatch_async function to execute a block of code in the main thread. But according to Apple, it is recommended to use NSOperationQueue over dispatch_async. As a general rule, Apple recommends using the highest-level APIs rather than dropping down to the low-level ones.

Displaying Loans in A Table View

With the loans array in place, the last thing we need to do is to display the data in the table view. Update the following methods in `KivaLoanTableViewController.swift`:

```
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    // Return the number of sections.
    return 1
}

override func tableView(tableView: UITableView, numberOfRowsInSection section:
Int) -> Int {
    // Return the number of rows in the section.
    return loans.count
}

override func tableView(tableView: UITableView, cellForRowAtIndexPath
indexPath: NSIndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("Cell",
forIndexPath: indexPath) as! KivaLoanTableViewCell

    // Configure the cell...
    cell.nameLabel.text = loans[indexPath.row].name
    cell.countryLabel.text = loans[indexPath.row].country
    cell.useLabel.text = loans[indexPath.row].use
```

```
    cell.amountLabel.text = "$\(loans[indexPath.row].amount)"

    return cell
}
```

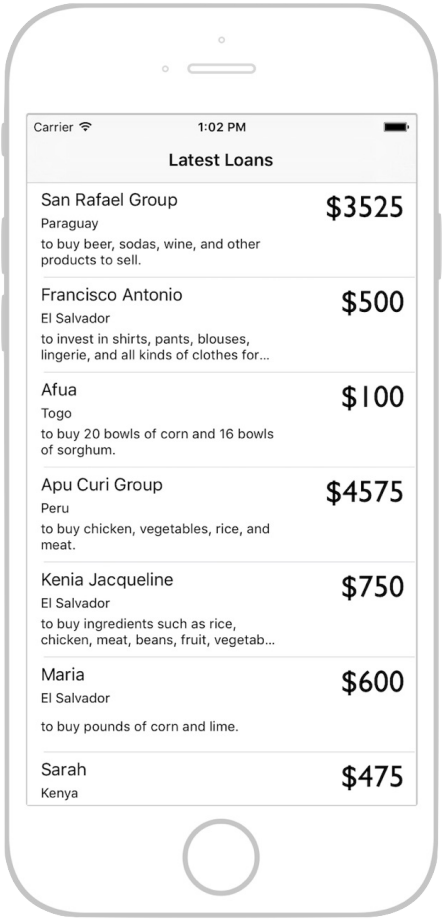
The above code is pretty straightforward if you are familiar with the implementation of `UITableView`. In the `cellForRowAtIndexPath:` method, we retrieve the loan information from the loans array and populate them in the custom table cell. One thing to take note of is the code below:

```
"$\(loans[indexPath.row].amount)"
```

Sometimes you may want create a string by adding both string (e.g. $) and integer (e.g. `loans[indexPath.row].amount`) together. Swift provides a powerful way to create these kinds of strings, known as string interpolation. You can make use of it by using the above syntax.

Lastly, insert the following line of code in the `viewDidLoad` method to start fetching the loan data:

```
getLatestLoans()
```



Compile and Run

Now it's time to test the app. Compile and run it in the simulator. Once launched, the app will pull the latest loans from Kiva.org and display them in the table view.

For your reference, you can download the complete Xcode project from https://www.dropbox.com/s/ze9f3n2q3tzu341/KivaLoan.zip?dl=0.