Public APIs

<https://github.com/public-apis/public-apis>

Spaceflight News API

<https://api.spaceflightnewsapi.net/v3/articles>

Spaceflight News API Documentation

<https://api.spaceflightnewsapi.net/v3/documentation#/Article>

Learning Git

<https://githowto.com/>

Charts Documentary

<https://weeklycoding.com/mpandroidchart-documentation/>

Chart Library

<https://github.com/danielgindi/Charts>

<https://cocoapods.org/pods/Charts>

How to create Charts in iOS Application

<https://medium.com/@raj.amsarajm93/swift-how-to-create-charts-in-ios-application-3c58efa98561>

CC AutoComplete Library

<https://cocoapods.org/pods/CCAutocomplete>

# iOS as a Front-End

A website can be accessed from any device with a browser. This is why a website is crucial for any company because of its accessibility. However, it is hard to argue against the fact that a native application provides a better user experience than a website can. Unfortunately, the browser cannot compete with the power of native applications that have the device's full power at its disposal. So we can build a website and a native application. However, we want them to share the database. The presentation of the content will be different for each device (the way the user interacts with the content) but the content itself is the same. To have the accessibility to the web, while harnessing the power of a native application, developers create APIs. By creating an API, other developers (including us) can come up with different ways to present the same data.

**Using iOS as a front-end means requesting data from a back-end server via an API using AJAX.**

Before we write our own back-end to provide data we'll use a third-party API to learn how to make AJAX requests in iOS.

Start by *creating a new project called "Star Wars Encyclopedia"  and***then deleting the default view controller and replacing it with a table view controller.**

**Be sure to do the following before moving on (refer back to the iOS intermediate section if necessary):**

* Delete the default view controller and replace it with a table view controller
* Rename the ViewController.swift file to PeopleViewController.swift and rename the class as well
* Make sure that PeopleViewController inherits from UITableViewController instead of UIViewController
* Set the Custom Class for the People View Controller Scene to be the PeopleViewController class.
* Give the Prototype Cell an identifier "personCell"

Now let's populate the PeopleViewController.swift class with the necessary methods to run our application. We'll be hard coding the data to start before we hook up to the third-party API.

### PeopleViewController.swift

import UIKit

class PeopleViewController: UITableViewController {

// Hardcoded data for now

var people = ["Luke Skywalker", "Leia Organa", "Han Solo", "C-3PO", "R2-D2"]

override func viewDidLoad() {

super.viewDidLoad()

// Do any additional setup after loading the view, typically from a nib.

}

override func didReceiveMemoryWarning() {

super.didReceiveMemoryWarning()

}

override func numberOfSections(in tableView: UITableView) -> Int {

// if we return - sections we won't have any sections to put our rows in

return 1

}

override func tableView(\_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {

// return the count of people in our data array

return people.count

}

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

// Create a generic cell

let cell = UITableViewCell()

// set the default cell label to the corresponding element in the people array

cell.textLabel?.text = people[indexPath.row]

// return the cell so that it can be rendered

return cell

}

}

And let's see our application work! Not too fancy huh? Let's add in the third-party API.

We'll be using the  [SWAPI](https://swapi.co/) for this assignment. It's a great API that contains a lot of awesome Star Wars information and it's free to use! Our goal is to replace our hard coded data with information from the SWAPI.

In order to get all of the people in the Star Wars universe, we'll have to make a request to  <https://swapi.dev/api/people/?format=json>. Go there now to see all of the data that we'll be getting back.

Here is a sample of the data that we'll be getting back:

{

"count": 87,

"next": "http://swapi.co/api/people/?page=2",

"previous": null,

"results": [

{

"name": "Luke Skywalker",

"height": "172",

"mass": "77",

"hair\_color": "blond",

"skin\_color": "fair",

"eye\_color": "blue",

"birth\_year": "19BBY",

"gender": "male",

"homeworld": "http://swapi.co/api/planets/1/",

"films": [

"http://swapi.co/api/films/6/",

"http://swapi.co/api/films/3/",

"http://swapi.co/api/films/2/",

"http://swapi.co/api/films/1/",

"http://swapi.co/api/films/7/"

],

"species": [

"http://swapi.co/api/species/1/"

],

"vehicles": [

"http://swapi.co/api/vehicles/14/",

"http://swapi.co/api/vehicles/30/"

],

"starships": [

"http://swapi.co/api/starships/12/",

"http://swapi.co/api/starships/22/"

],

"created": "2014-12-09T13:50:51.644000Z",

"edited": "2014-12-20T21:17:56.891000Z",

"url": "http://swapi.co/api/people/1/"

},

{

"name": "C-3PO",

"height": "167",

"mass": "75",

"hair\_color": "n/a",

"skin\_color": "gold",

"eye\_color": "yellow",

"birth\_year": "112BBY",

"gender": "n/a",

"homeworld": "http://swapi.co/api/planets/1/",

"films": [

"http://swapi.co/api/films/5/",

"http://swapi.co/api/films/4/",

"http://swapi.co/api/films/6/",

"http://swapi.co/api/films/3/",

"http://swapi.co/api/films/2/",

"http://swapi.co/api/films/1/"

],

"species": [

"http://swapi.co/api/species/2/"

],

"vehicles": [],

"starships": [],

"created": "2014-12-10T15:10:51.357000Z",

"edited": "2014-12-20T21:17:50.309000Z",

"url": "http://swapi.co/api/people/2/"

},

...

Wow, that's a lot of info! This basically tells us that if we make a GET request to swapi.co/api/people we will get this large json object back as a response. **Hypothetically if we saved that large JSON object into a variable called "res" how would we access the first person (Luke Skywalker)?**

1. Access the "results" key on the "res" object
2. The first entry in the array corresponding to the "results" key is an object that represents Luke Skywalker
3. The name key in this object holds "Luke Skywalker" as a value.

## Great now how do we actually make the request?

### App Transport Security Settings and Info.plist

Info.plist stands for "Information Property List". The Info.plist file is a structured text file that contains configurations and settings for your application. The text format that info.plist uses is XML but for our purposes, we will simply interact with the file through the XCode interface. We will be modifying the Info.plist file to let our application know that we want to be able to request data via HTTP. By default, XCode's Transport Security policy will block these requests because they are insecure.

First, go into info.plist and press the "+" icon next to "Information Property List". Add the "App Transport Security Settings" key like so:

صورة تحتوي على نص

تم إنشاء الوصف تلقائياً

Then click on the "+" icon in App Transport Security Settings and add the "Allow Arbitrary Loads" key like so:

صورة تحتوي على نص

تم إنشاء الوصف تلقائياً

Finally set "Allow Arbitrary Loads" to "YES" like so:

صورة تحتوي على نص

تم إنشاء الوصف تلقائياً

And now we have configured our application to be able to request data via HTTP! Let's get back to creating our request. For every application where you want to make HTTP requests you will need to configure these settings. In our case we have "Allowed Arbitrary Loads" which means we can make an HTTP request to any server and receive data back from any server. **To be more secure in your application you can set "Allow Arbitrary Loads" to "NO" and then set specific domains in the "Exception Domains" key.** For this application we will leave "Allow Arbitrary Loads" as "Yes".

### Our First Request

For now, we will make the request in our viewDidLoad method for simplicity. Later we will show how you can modularize your code into more of an MVC type architecture but for now, we'll put the HTTP request in the viewDidLoad function so that we can pull the data from the API as soon as the view loads.

Let's add the following code to our viewDidLoad function to make a request:

override func viewDidLoad() {

super.viewDidLoad()

// specify the url that we will be sending the GET Request to

let url = URL(string: "https://swapi.dev/api/people/?format=json")

// create a URLSession to handle the request tasks

let session = URLSession.shared

// create a "data task" to make the request and run the completion handler

let task = session.dataTask(with: url!, completionHandler: {

// see: Swift closure expression syntax

data, response, error in

print("in here")

// see: Swift nil coalescing operator (double questionmark)

print(data ?? "no data") // the "no data" is a default value to use if data is nil

})

// execute the task and wait for the response before

// running the completion handler. This is async!

task.resume()

}

Here is a breakdown of the code above:

First, we created an URL object by passing in the string for the URL that we wanted to request.

let url = URL(string: "https://swapi.dev/api/people/?format=json")

Then we created a URLSession that we can use to run "tasks" to transfer data over HTTP.

let session = URLSession.shared

Next, we set up a particular task as a "data task". Running the "session.dataTask" method will request some data from the given URL and then run the completionHandler code block as soon as the data response is received. The double questionmark is called a "nil coalescing operator" and is used to provide an alternative argument in the case that an optional is nil.

let task = session.dataTask(with: url!, completionHandler: {

data, response, error in

print("in here")

print(data ?? "no data")

})

Finally, we actually execute the task by running the resume method on our task object.

task.resume()

**Now run the program and take a look at the data that we printed in the console. Pretty disappointing huh?**

### JSON data and Swift

When we printed the "data" variable in the code above we saw a bunch of gibberish in the console. One of the toughest parts of learning Swift and iOS is learning how to deal with JSON data. In Swift, we have several types to identify dictionaries and arrays, but it is tough to map these collection types to JSON data that is received via an HTTP request. In the next chapter, we will explore the tools at our disposal to "deserialize" the JSON data into a format that is usable in Swift.

**JSON Data in Swift**

In the last chapter, we learned how to request some data via HTTP. In this chapter, we will learn how to process that JSON data into types that can be used in Swift. Remember that Swift is strongly typed and, therefore, everything that we interact with must have a specific type. In order to handle JSON data we can use a library called "JSONSerialization". Let's see this in action.

Add the following code to your HTTP request in the viewDidLoad method of your PeopleViewController class.

override func viewDidLoad() {

super.viewDidLoad()

// specify the url that we will be sending the GET request to

let url = URL(string: "https://swapi.dev/api/people/?format=json")

// create a URLSession to handle the request tasks

let session = URLSession.shared

// create a "data task" to make the request and run completion handler

let task = session.dataTask(with: url!, completionHandler: {

// see: Swift closure expression syntax

data, response, error in

// data -> JSON data, response -> headers and other meta-information, error-> if one occurred

// "do-try-catch" blocks execute a try statement and then use the catch statement for errors

do {

// try converting the JSON object to "Foundation Types" (NSDictionary, NSArray, NSString, etc.)

if let jsonResult = try JSONSerialization.jsonObject(with: data!, options: JSONSerialization.ReadingOptions.mutableContainers) as? NSDictionary {

print(jsonResult)

}

} catch {

print(error)

}

})

// execute the task and then wait for the response

// to run the completion handler. This is async!

task.resume()

}

Let's examine the code inside of the completionHandler. The most important line is:

if let jsonResult = try JSONSerialization.jsonObject(with: data!, options: JSONSerialization.ReadingOptions.mutableContainers) as? NSDictionary {

We use the JSONSerialization library to attempt to convert the data object into JSON, and then unwrap it using our standard optional unwrapping syntax while typecasting the jsonResult object to an NSDictionary. With Swift 5, the NS prefix has been mostly done-away with, but we \*do\* have to use the NS prefix here.

Try running the code and now look at your console to see the JSON data!

Now how would we access the first element inside of the "results" key? First, let's try to drill down into the results array itself.

if let jsonResult = try JSONSerialization.jsonObject(with: data!, options: JSONSerialization.ReadingOptions.mutableContainers) as? NSDictionary {

// Why do we need to optionally unwrap jsonResult["results"]

// Try it without the optional unwrapping and you'll see that the value is actually an optional

if let results = jsonResult["results"] {

print(results)

}

}

**Typecasting**

If you option-click on the "results" object in the code above, you will see that the result is an instance of "Any". What if we wanted to run specific methods on the results object and treat it as an Array? We would have to typecast the Any as an NSArray type. This is called Type Coercion or Downcasting. Let's see an example of this:

if let jsonResult = try JSONSerialization.jsonObject(with: data!, options: JSONSerialization.ReadingOptions.mutableContainers) as? NSDictionary {

// Why do we need to optionally unwrap jsonResult["results"]

// Try it without the optional unwrapping and you'll see that the value is actually an optional

if let results = jsonResult["results"] {

// coercing the results object as an NSArray and then storing that in resultsArray

let resultsArray = results as! NSArray

// now we can run NSArray methods like count and firstObject

print(resultsArray.count)

print(resultsArray[0])

print(resultsArray.firstObject)

}

}

There may be many times that you have to force downcast an AnyObject to a specific type. When dealing with JSON data we advise that you use the following types:

* NSArray
* NSMutableArray
* NSDictionary
* NSMutableDictionary

We advise using these types because they are part of the Foundation framework available in iOS and using some of the default Swift types like Array and Dictionary may cause problems when working with JSONSerialization (since JSONSerialization comes from the Foundation framework as well).