raywenderlich.com

Swift JSON Tutorial: Working with JSON

Luke Parham on February 20, 2017

Check out the video for this tutorial here

Note: Updated for Xcode 8.2 and Swift 3 on 1-15-2017 by Luke Parham.
The original tutorial was written by Attila Hegedüs.

JavaScript Object Notation, or **JSON** for short, is a common way to transmit data to and from web services. It's simple to use and human-readable, which is why it's so incredibly popular.

Consider the following JSON snippet:

```
[
  {
    "person": {
      "name": "Dani",
      "age": "24"
    }
  },
  {
    "person": {
      "name": "Ray",
      "age": "70"
    }
  }
]
```

In **Objective-C**, parsing and deserializing JSON is fairly straightforward:

```
NSArray *json = [NSJSONSerialization JSONObjectWithData:JSONData options:kNilOptions error:nil];
NSString *age = json[0][@"person"][@"age"];
NSLog(@"Dani's age is %@", age);
```

In **Swift**, parsing and deserializing JSON is a little more tedious due to optionals and type-safety.

```
var json: [Any]?
do {
  json = try JSONSerialization.jsonObject(with: data)
} catch {
  print(error)
```

```
}
guard let item = json?.first as? [String: Any],
  let person = item["person"] as? [String: Any],
  let age = person["age"] as? Int else {
    return
}
```

While the **guard** statement does make it so your JSON parsing logic no long requires you to build up your own pyramid of doom, you still need to write a bit of boilerplate to get at the data in a JSON string.

In this Swift JSON tutorial, you'll start by parsing JSON the native Swift way – i.e. without using any external libraries. This is actually the strategy Apple recommends since the built-in tools available in Swift should technically be sufficient for most use cases.

That being said, everyone feels the need to rebel every now and then, so you'll spend the second half of this Swift JSON tutorial learning how quickly things can go by using a framework called Gloss.

> **Note:** Gloss is just one of the many JSON frameworks you can choose from, but its a good example of patterns you'll see in a lot of them.

You'll use these two strategies to parse a JSON document containing the top 25 most popular apps in the US App Store.

Getting Started

Since a user interface isn't necessary to learn about JSON, you'll do all of your work in a playground. Download the starter playground here.

Open **Swift.playground** in Xcode and have a look around.

> **Note:** You may find the playground Project Navigator is closed by default. If so, simply press ⌘+1 to display it.

The starter playground file has a few source and resource files included to keep the focus purely on parsing JSON in Swift. Take a look at its structure to get an overview of what's included:

- The **Resources** folder is where you'll find your sample JSON file to be parsed.
  - **topapps.json**: Contains the JSON string itself.
- The **Sources** folder contains additional Swift source files your main playground code can access. Putting extra supporting **.swift** files into the Sources folder makes it easy to keep your playground clean and readable.
  - **App.swift**: A plain old Swift struct representing an app. Your goal is to parse the JSON into a collection of these objects.
  - **DataManager.swift**: Manages the data retrieval whether local or from the network. You'll use the methods in this file to load some JSON later in this Swift JSON tutorial.

Once you feel like you've a good understanding of what's currently in the playground, read on!

Parsing JSON the Native Swift Way

Now you've seen where the sample JSON file is, its time to parse it and print out the name of the #1 app in the App Store!

Using an Optional Initializer

First, open **App.swift**. Here, you'll find a simple model object with an initializer that takes a name and link.

The JSON file you're dealing with has a list of objects under the key **"entry"**, each of which can be represented by an **App** object.

To accommodate this structure, add the following optional initializer below the original one.

```
//1
public init?(json: [String: Any]) {
  //2
  guard let container = json["im:name"] as? [String: Any],
    let name = container["label"] as? String,
    let id = json["id"] as? [String: Any],
    let link = id["label"] as? String else {
      return nil
  }
  //3
  self.name = name
  self.link = link
}
```

1. This initializer takes a dictionary of type **[String: Any]** which has been constructed from the JSON data. While this is correct; valid JSON can also be an array or just a plain value, however we know ours follows this dictionary pattern.
2. Next, use **guard-let** to dig into the JSON a little bit and pull out the **name** and **link** values.
3. Finally, if none of the values were **nil**, the values are set on the model object's properties and the model is successfully returned.

Now you have your model set up, open the main playground file by clicking on **TopApps-Starter** in the project navigator.

First, add the following **guard** statement inside the success callback of **getTopAppsDataFromFileWithSuccess(_:)** method.

```
guard let json = try? JSONSerialization.jsonObject(with: data) as? [String: Any] else {
  PlaygroundPage.current.finishExecution()
}
```

Here you use the **jsonObject(with:options:)** method to try to convert the JSON string to a JSON object.

Next, you'll do a little up front parsing to get to the JSON representing the first **App** object.

```
guard let feed = json?["feed"] as? [String: Any],
  let apps = feed["entry"] as? [[String: Any]],
  let firstApp = apps.first else {
    PlaygroundPage.current.finishExecution()
}
```

Here you grab the outermost object named **"feed"**, which contains an array of apps under the **"entry"** key.

Finally, use the optional initializer you wrote to retrieve the first app by adding the following:

```
let app = App(json: firstApp)
print(app ?? "Failed to initialize")
```

Build to see the following result in the console:

```
App(name: "Game of War - Fire Age", link: "https://itunes.apple.com/us/app/game-of-war-fire-age/id667728512?mt=8&uo=2")
```

Yes — “Game of War – Fire Age” is the #1 app in the JSON file.

Using an Initializer with Error Handling

While having the initialization fail by returning **nil** instead of an initialized object is one way to go about things, it really doesn’t tell you much about what went wrong.

Open **App.swift** and define the following error **enum** at the top of the file.

```
enum SerializationError: Error {
  case missing(String)
}
```

Next, replace the optional initializer you wrote earlier with the following initializer to the struct definition. This initializer throws the error you just defined when an expected value **nil** inside the provided JSON.

```
public init(json: [String: Any]) throws {
  //1
  guard let container = json["im:name"] as? [String: Any],
    let name = container["label"] as? String else {
      throw SerializationError.missing("name")
  }
  guard let id = json["id"] as? [String: Any],
    let link = id["label"] as? String else {
      throw SerializationError.missing("link")
  }
  //2
  self.name = name
  self.link = link
}
```

1. Here, you set up a separate guard for each of your properties. Next, if the **JSON** passed in was invalid you throw an error specifying which of the properties couldn’t be found.
2. Finally, you set the properties and return a valid object if nothing went wrong.

Open the main playground and replace

```
let app = App(json: firstApp)
print(app ?? "Failed to initialize")
```

with the following:

```
do {
  let app = try App(json: firstApp)
  print(app)
} catch let error {
  print(error)
}
```

Here, you use a do-catch block. If the initialization is successful you don’t have to worry about optionals. When it fails you get a helpful error instead of a **nil** object.

To see your error in action, just replace **App(json: firstApp)** with **App(json: [:])**.

Parsing JSON the Gloss way

Now you've some experience mapping JSON to your own model objects, its time to take a look at the alternative.

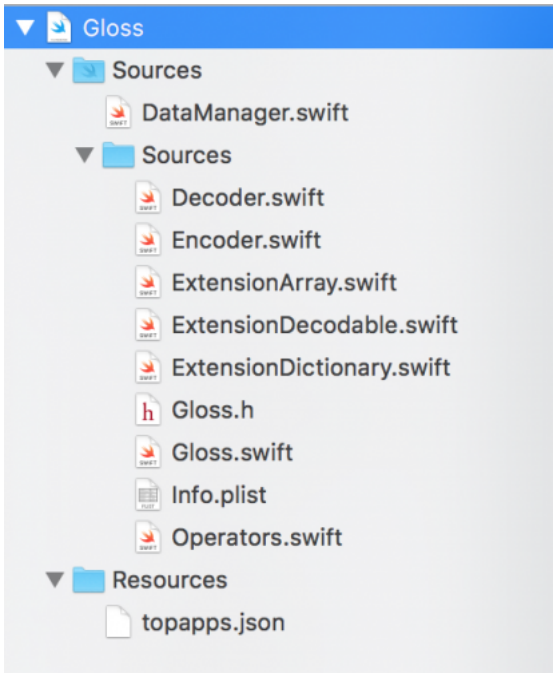
To keep everything nice and clean, create a new playground called **Gloss.playground**. Next, copy over **topapps.json** into **Resources** and **DataManager.swift** into **Sources**.

Integrating Gloss in Your Project

It's fairly straightforward to integrate **Gloss** into your project/playground:

1. Download the **Gloss** repo zip file here.
2. Unzip and drag the **Gloss-master/Sources** folder into the **Sources** folder of your playground.

Your **Project Navigator** should look like the following:



That's it! Now you've added **Gloss** to your playground, you can start parsing JSON the "easy" way!

Mapping JSON to objects

First, you have to define your model objects along with how they relate to your JSON object.

Model objects must conform to the **Decodeable** protocol, which enables them to be decoded from JSON. To do so, you'll implement the **init?(json: JSON)** initializer.

> **Note**: Open **Gloss.swift**. Inside the **Decodable** protocol, if you ⌘+click **JSON** to look at its definition you'll see it's just a **typealias** for **[String: Any]** defined in the same file.

TopApps

The **TopApps** model represents the top level object, and contains a single key-value pair:

```
{
  "feed": {
```

```
    ...
  }
}
```

Create a new Swift file called **TopApps.swift** in the **Sources** folder of your playground and add the following code:

```
public struct TopApps: Decodable {

  // 1
  public let feed: Feed?

  // 2
  public init?(json: JSON) {
    feed = "feed" <~~ json
  }

}
```

1. First you define your model's properties. In this case there's only one. Don't worry about the **Feed** error, you'll define the **Feed** model object next.
2. **TopApps** is required to implement the optional initializer from before in order to conform to the **Decodable** protocol.

You might be wondering what **<~~** is! It's called the **Encode Operator** and it's defined in **Gloss**'s **Operators.swift** file. Basically it tells **Gloss** to grab the value which belongs to the key **feed** and **encode** it. **Feed** is a **Decodable** object as well; so **Gloss** will delegate the responsibility of the encoding to this object.

Feed

The **Feed** object is very similar to the top level object. **Feed**s have two key-value pairs, but since you're only interested in the top 25 apps, it's not necessary to process the **author** object.

```
{
  "author": {
    ...
  },
  "entry": [
    ...
  ]
}
```

Create a new Swift file called **Feed.swift** in the **Sources** folder of your playground and define **Feed** as follows:

```
public struct Feed: Decodable {

  public let entries: [App]?

  public init?(json: JSON) {
    entries = "entry" <~~ json
  }

}
```

App

App is the last model object you are going to define. It represents an app in the chart:

```
{
  "im:name": {
```

```
    "label": "Game of War - Fire Age"
  },
  "id": {
    "label": "https://itunes.apple.com/us/app/game-of-war-fire-age/id667728512?mt=8&uo=2",
    ...
  },
  ...
}
```

Create a new Swift file called **App.swift** in the **Sources** folder of your playground and add the following code:

```
public struct App: Decodable {

  // 1
  public let name: String
  public let link: String

  public init?(json: JSON) {
    // 2
    guard let container: JSON = "im:name" <~~ json,
      let id: JSON = "id" <~~ json else {
        return nil
    }

    guard let name: String = "label" <~~ container,
      let link: String = "label" <~~ id else {
        return nil
    }

    self.name = name
    self.link = link
  }
}
```

1. **Feed** and **TopApps** both used **optional properties**. Instead you can define a property as **non-optional** if you're sure the JSON being used will always contain the values to populate them.
2. You don't necessarily have to create model objects for every member in the JSON. For example, in this case it makes no sense to create a model for **in:name** and **id**. When working with non-optional and nested objects, always make sure to check for **nil**.

Now your model classes are ready the only thing left is to let **Gloss** do it's job! :]

Open the playground file and replace its contents with the following:

```
import UIKit
import PlaygroundSupport

PlaygroundPage.current.needsIndefiniteExecution = true
URLCache.shared = URLCache(memoryCapacity: 0, diskCapacity: 0, diskPath: nil)

DataManager.getTopAppsDataFromFileWithSuccess { (data) -> Void in

  // 1
  var json: Any
  do {
    json = try JSONSerialization.jsonObject(with: data)
  } catch {
    print(error)
    PlaygroundPage.current.finishExecution()
```

```
  }

  guard let dictionary = json as? [String: Any] else {
    PlaygroundPage.current.finishExecution()
  }

  // 2
  guard let topApps = TopApps(json: dictionary) else {
    print("Error initializing object")
    PlaygroundPage.current.finishExecution()
  }

  // 3
  guard let firstItem = topApps.feed?.entries?.first else {
    print("No such item")
    PlaygroundPage.current.finishExecution()
  }

  print(firstItem)

  PlaygroundPage.current.finishExecution()
}
```

1. First you deserialize the data using **JSONSerialization** just like before.
2. Next, initialize an instance of **TopApps** by feeding the JSON data into it's constructor.
3. Finally, grab the first app by using the **feed** and **entries** properties on the model objects you created.

That's all there is to it!

Save again to see you successfully get the app name again, but this time in a more elegant way:

```
App(name: "Game of War - Fire Age", link: "https://itunes.apple.com/us/app/game-of-war-fire-age/id667728512?mt=8&uo=2")
```

> **Note:** If at this point you get an error that says **topapps.json** couldn't be opened because you don't have permission, make sure the starter playground is closed, then delete the contents of your derived data folder.

That takes care of parsing local data — but how would you parse data from a remote source?

Retrieving Remote JSON

It's time to make this project a bit more realistic. Normally, you'd be retrieving remote data, not data from a local file. You can easily grab the App Store ratings using a network request.

Open **DataManager.swift** and define **topAppURL** just before the **DataManager**'s implementation:

```
let topAppURL = "https://itunes.apple.com/us/rss/topgrossingipadapplications/limit=25/json"
```

Next, add the following method to **DataManager**'s implementation:

```
public class func getTopAppsDataFromItunesWithSuccess(success: @escaping ((_ iTunesData: Data) -> Void)) {
  //1
  loadDataFromURL(url: URL(string: topAppURL)!) { (data, error) -> Void in
```

```
    //2
    if let data = data {
      //3
      success(data)
    }
  }
}
```

The above code looks pretty familiar; but instead of retrieving a local file you're using **URLSession** to pull the data from **iTunes**. Here's what's happening in detail:

1. First you call **loadDataFromURL**; this takes the URL and a completion closure that passes in a **Data** object.
2. Next, you make sure the data exists by using optional binding.
3. Finally, you pass the data to the success closure as you did before.

Open the main playground file and replace

```
DataManager.getTopAppsDataFromFileWithSuccess { (data) -> Void in
```

with the following:

```
DataManager.getTopAppsDataFromItunesWithSuccess { (data) -> Void in
```

Now you're retrieving real data from **iTunes**.

Save one last time and you'll see whatever the current hottest game is. For me it was still **Candy Crush Saga**. Really crushin' it out there **Candy Crush**.

```
App(name: "Candy Crush Saga", link: "https://itunes.apple.com/us/app/candy-crush-saga/id553834731?mt=8&uo=2")
```

The above value may be different for you, as the top apps in the App Store change constantly.

Usually people aren't just interested in the top app in the App Store — they want to see a list of all top apps. You don't have to code anything in order to access them - you can easily access them with the following code snippet:

```
topApps.feed?.entries
```

Peeking Under the Hood at Gloss

As you saw, **Gloss** works really well when it comes to parsing JSON data — but how does it work under the hood?

<~~ is a **custom operator** for a set of **Decoder.decode** functions. Gloss has built-in support for decoding a lot of types:

- Simple types (**Decoder.decode(key:)**)
- Decodable models (**Decoder.decode(decodableForKey:)**)
- Simple arrays (**Decoder.decode(key:)**)
- Arrays of Decodable models (**Decoder.decode(decodableArrayForKey:)**)
- Enum types (**Decoder.decode(enumForKey:)**)
- Enum arrays (**Decoder.decode(enumArrayForKey:)**)
- URL types (**Decoder.decode(urlForKey:)**)

- URL arrays (**Decode.decode(urlArrayForKey:)**)

In this Swift JSON tutorial you relied heavily on **Decodable** models. If you need something more complex, it's possible to extend **Decoder** and provide your own decoding implementation.

Of course **Gloss** can convert objects back to JSON as well. If you're interested in doing that, have a look at the **Encodable** protocol.

Where to Go From Here?

Here's the final playground from the above Swift JSON tutorial.

If you preferred the Apple way of doing things in the first half of this Swift JSON tutorial, you might like their full blog post on using JSON with Swift. Their final example of cleanly building up a network layer that deals with JSON is especially useful.

Alternatively, if you preferred the **Gloss** approach, development will continue on Github, so keep an eye out for the latest updates.

I hope you enjoyed this Swift JSON tutorial; don't forget to check out all the other Swift tutorials on the website. If you have any questions or comments, feel free to join the discussion below!

Team

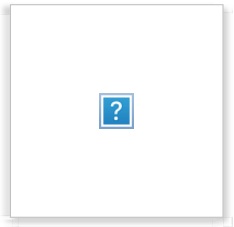
Each tutorial at www.raywenderlich.com is created by a team of dedicated developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Author
Luke Parham

Tech Editor
Michael Gazdich

Final Pass Editor
Darren Ferguson

Team Lead
Andy Obusek



Luke Parham

Luke is currently an iOS developer at Fyusion.
Sometimes he tweets and blogs.

He basically speaks in references so if you catch one be sure to let him know.