ATLS 4320: Advanced Mobile Application Development
Week 6: JSON

JSON

JavaScript Object Notation (JSON) is a language independent, human-readable data format used for transporting data between two systems. http://json.org/
Supported by every major modern programming language including JavaScript, Swift, and Java. JSON has a limited number of data types: string, boolean, array, object/dictionary, null and number.
JSON is built on two structures

- A collection of name/value pairs stored as an object, record, struct, dictionary, hash table, keyed list, or associative array in various languages
  - An object in in curly brackets { }
  - Format: name : value
  - Name/value pairs are separated by a comma ,
- An ordered list of values are stored as an array, vector, list, or sequence in various languages
  - An array is in square brackets [ ]
  - Values are separated by a comma

JSON:
https://www.googleapis.com/youtube/v3/search?key=AIzaSyB9sA6hpuTZ8rOOAh5z5A3tcnSMPP2JUJA&part=snippet&maxResults=10&q=cats
To display the JSON formatted in Chrome you'll need to install an extension.

iOS Networking

There are three main classes you need to know about in order to handle networking in iOS:

1. URLRequest encapsulates information about a URL request.
   https://developer.apple.com/documentation/foundation/urlrequest
   - Can specify the url of the request, http method, headers, timeout interval, etc
   - Used by URLSession to send the request
2. URLSession coordinates the set of requests and responses that make up a HTTP session
   https://developer.apple.com/documentation/foundation/urlsession
   - URLSession has a singleton shared session for basic requests
     https://developer.apple.com/documentation/foundation/urlsession/1409000-shared
   - URLSession.shared.dataTask(with:completionHandler:) returns a URLSessionTask instance with the contents of a URL
     https://developer.apple.com/documentation/foundation/urlsession/1407613-datatask
     i. Requests a URLRequest
     ii. Completion handler to call when the request is complete and successful
         1. Data – holds the downloaded data if successful
         2. Response - An object that provides response metadata, such as HTTP headers and status code. If you are making an HTTP or HTTPS request, the returned object is actually an HTTPURLResponse object.
            a. The HTTP status code is stored in the HTTPURLResponse statusCode property
            b. HTTP status codes
               i. https://en.wikipedia.org/wiki/List_of_HTTP_status_codes
               ii. 200 is OK
         3. Error – an error object if the request fails, data will be nil
     iii. After you create the task, you must start it by calling its resume method

iv. Returns the new session data task
v. URLSessionTask performs the actual transfer of data through the URLSession instance. It has different subclasses for different types of tasks. URLSessionDataTask is used for the contents of an URL.

In summary, making an HTTP request in iOS:

- create and configure an instance of URLSession, to make one or more HTTP requests
- optionally create and configure an instance URLRequest for your requests. Needed only if you need some specific parameters.
- start a URLSessionDataTask through the URLSession class.

JSON in iOS
Once the JSON has been downloaded successfully we need to parse the data and save it in our data model

- The DispatchQueue class manages the execution of work items. Each work item submitted to a queue is processed on a pool of threads managed by the system.
- The DispatchQueue.main.async{} method will submit requests to be run asynchronously on another threads. You should use this to parse the JSON asynchronously. It's really important to only use the main queue for the UI, otherwise the other tasks make the app unresponsive and slow as it's waiting on the other tasks.
- Instead of the PropertyListDecoder we've been using for plists, we'll use the JSONDecoder which is the JSON equivalent. Just as with property lists, we need the property names of our struct to match the key name in the JSON file.

jokes
iOS App
Product Name: jokes
Team: None
Org identifier: ATLAS (can be anything, will be used in the bundle identifier)
Interface: Storyboard
Life Cycle: UIKit App Delegate
Language: Swift
Uncheck core data and include tests.
Uncheck create local git repo

<u>Setup</u>
Go into MainStoryboard.
Add an image view at the top of the view.
Add an image to your project and assign it to the image view.
Add a table view that approximately covers the bottom 2/3 of the view.
Select the table view and in the connections inspector connect the dataSource and delegate to the view controller icon (self).
You also need to connect the table view as an outlet to our ViewController class called tableView.
You'll also need to add a table view cell from the object library. Make sure it's added in the table view.
Select the Table View Cell and in the attributes inspector change the Style to Basic and add an identifier "jokeIdentifier".
For constraints I embedded these both in a stack view and added constraints to all sides with the value of 10. Alignment and distribution are both fill.

In our ViewController class because we subclass UIViewController and not UITableViewController, we need to specifically adopt the protocols for table views.

```
class ViewController: UIViewController, UITableViewDelegate,
UITableViewDataSource
```

You'll then get an error about conforming to those protocols and you can select to add the stub methods.

Data
You have to carefully look at JSON data to understand its structure and set up a structure that matches it so we can use the JSONDecoder decode method for the data from the API.
Documentation: https://dadjokes.io/documentation/endpoints/random-jokes
https://rapidapi.com/KegenGuyll/api/dad-jokes?endpoint=apiendpoint_d9018ef4-ad90-4037-b8fa-4ff5a766e234
You can look at the example responses and the results to see the structure.
An object is returned with key/value pairs.
The first key is "body" and the second is "success".
The value for "body" is an array. Each item in the array is an object with key/value pairs. Keys:

- _id
- punchline
- setup
- type

I only need the setup and the punchline so that's what I'll include in my struct.
Note that you must use these keys exactly in your struct for the decoder to work.

Data Model
We're going to use a struct to represent a joke. We'll pick out the data items in the value of body that we're interested in.
File | New | File | Swift File
Joke

```
struct Joke: Decodable {
    let setup: String
    let punchline: String
}
```

To use a JSONDecoder we need a data structure with a property named body and the value an array.

```
struct JokeData: Decodable {
    var body = [Joke]()
}
```

In ViewController.swift we'll need an array of jokes to use for our table view.

```
    var jokes = [Joke]()
```

Now let's get the table view set up by implementing the data source and delegate methods.

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

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: 
IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: 
"jokeIdentifier", for: indexPath)
        let joke = jokes[indexPath.row]
        cell.textLabel!.text = joke.setup
        return cell
    }
```

If you run it at this point you should see the table but it will be empty since we need the joke data.

Test Data

Before we download the JSON data it's not a bad idea to get the rest of the app working with some test data, especially if the API you're using has limited free results.

In ViewController I'm going to create a method to create some test data that we can use. (This can also go in your data handler class which we will create shortly)

```
    func loadtestdata() {
        //test data
        let joke1 = Joke(setup: "What's the best thing about a Boolean?", 
punchline: "Even if you're wrong, you're only off by a bit.")
        let joke2 = Joke(setup: "What's the object-oriented way to become 
wealthy?", punchline: "Inheritance")
        let joke3 = Joke(setup: "If you put a million monkeys at a million 
keyboards, one of them will eventually write a Java program?", punchline: 
"the rest of them will write Perl")
        jokes.append(joke1)
        jokes.append(joke2)
        jokes.append(joke3)
    }
```

And then call it from viewDidLoad()

```
    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view.
        //jokeDataHandler.loadjson()
        //jokes = jokeDataHandler.getJokes()
        loadtestdata()
    }
```

Run the app and you should see the jokes in the table. But some of the jokes with longer text is getting cut off.

In the storyboard open the table view cell content view and select the label (default says title). In the attributes inspector change lines to 0 and Line Break to Word Wrap so the cell will grow based on the content.

Run it again and you should be able to see all the text. This is why it makes sense to work through these issue without using our limited API calls.

Now we need the punchlines for the jokes. Instead of a whole other view controller I thought an alert would work for this.

```
    func tableView(_ tableView: UITableView, didSelectRowAt indexPath: 
IndexPath) {
        let alert = UIAlertController(title: jokes[indexPath.row].setup, 
message: jokes[indexPath.row].punchline, preferredStyle: .alert)
        let okAction = UIAlertAction(title: "Haha", style: .default, 
handler: nil)
        alert.addAction(okAction)
        present(alert, animated: true, completion: nil)
        tableView.deselectRow(at: indexPath, animated: true) //deselects the 
row that had been choosen
    }
```

Now you should be able to select a joke and see both the setup and the punchline in the alert.
Now let's get the API call and JSON data working.

Load JSON file
Now we'll add a class to download our data and handle an instance of our data model.
File | New | File | Swift File
JokeDataHandler

```
class JokeDataHandler {
    var jokeData = JokeData()

    func loadjson(){
     //based on API documentation
        let headers = [
            "x-rapidapi-key": 
"4ed30fc463mshe74ccdc811abd17p194096jsn5a51929bf13d",
            "x-rapidapi-host": "dad-jokes.p.rapidapi.com"
        ]

        let urlPath = "https://dad-
jokes.p.rapidapi.com/joke/type/programming"

        guard let url = URL(string: urlPath)
            else {
                print("url error")
                return
            }

        var request = URLRequest(url: url, cachePolicy: 
.useProtocolCachePolicy, timeoutInterval: 10.0)

        request.httpMethod = "GET"
        request.allHTTPHeaderFields = headers

        let session = URLSession.shared.dataTask(with: request, 
completionHandler: {(data, response, error) in
            let httpResponse = response as! HTTPURLResponse
            let statusCode = httpResponse.statusCode
            print(statusCode)
            guard statusCode == 200
                else {
                    print("file download error")
                    return
                }
            //download successful
            print("download complete")
        })
        //must call resume to run session
        session.resume()
    }
```

Parse JSON file

Now we'll create a method to get the json data and parse it so we can use it.

```
    func parsejson(_ data: Data){
        do
        {
            let apiData = try JSONDecoder().decode(JokeData.self, from:
data)
            for joke in apiData.body
            {
                jokeData.body.append(joke)
            }
            print(jokeData.body.count)
        }
        catch let jsonErr
        {
            print("json error")
            print(jsonErr.localizedDescription)
            return
        }
        print("parsejson done")
    }

    func getJokes() -> [Joke] {
        return jokeData.body
    }
}
```

Call this asynchronously from loadjson() right after the print statement that the download was successful.

```
DispatchQueue.main.async {self.parsejson(data!)}
```

In ViewController.swift we'll update viewDidLoad() to call the loadjson() method and populate our jokes array. Remember to comment out loading the test data if you were using that.

```
    override func viewDidLoad() {
        super.viewDidLoad()
        jokeDataHandler.loadjson()
        jokes = jokeDataHandler.getJokes()
    }
```

If you run it now you'll probably see an empty table view. But if you had the debugging print statements in your code, you'll see the data was downloaded, so what happened?
Although we're calling loadjson() as early as we can, in viewDidLoad(), and getJokes() after that, these instructions don't really get called asynchronously, so getJokes() is completing before the JSON data is downloaded, so 0 jokes are returned. When the table view datasource methods are called there are therefore no jokes in the array to display in the table.
So we need a way to know when the data has been downloaded and parsed so we can reload our table.

Callback
There are three ways to pass data from a model to a controller:
https://medium.com/@stasost/ios-three-ways-to-pass-data-from-model-to-controller-b47cc72a4336

- Callbacks
- Delegation
- Notifications

Callbacks are a way to call some code, like a function, *after* another piece of code has finished.
Callbacks can be functions, or properties, that take the form of a closure.
A closure is basically a self-contained blocks of functionality that can be passed around. It can be a function but is usually unnamed (similar to anonymous functions or lambdas).

In JokeDataHandler.swift define the property onDataUpdate as a callback so once all the JSON data has been downloaded and parsed we pass this callback the results.

```
var onDataUpdate: ((_ data: [Joke]) -> Void)?
```

In parsejson() we pass the results to the callback once the json has been parsed.

```
        //passing the results to the onDataUpdate closure
        onDataUpdate?(jokeData.body)
```

In ViewController add a method that will get the list of jokes and reload the table data so the cells are populated with the data.

```
    func render() {
        jokes = jokeDataHandler.getJokes()
        tableView.reloadData()
    }
```

Then in viewDidLoad() we assign the onDataUpdate callback in the JokeDatahandler class a closure which is the render method.

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

        //assign the closure with the method we want called to the
onDataUpdate closure
```

```
        jokeDataHandler.onDataUpdate = {[weak self] (data:[Joke]) in
self?.render()}
        jokeDataHandler.loadjson()
      }
```

Why “weak”?
If you remember back to our discussion of Automatic Reference Counting (ARC), a weak reference does not keep a strong hold on the instance it refers to, and so does not stop ARC from disposing of the referenced instance. Closures are reference types so when you assign a closure to a property, you are assigning a strong reference to that closure. We don’t want to create another strong reference to the closure because the class already has a strong reference it and that would create a strong reference cycle. So we use a capture list (it’s an array as there could be more than one pairing) to define it as a weak reference since the property is an optional and may become `nil` at some point in the future. Weak references are always of an optional type, and automatically become `nil` when the instance the reference is deallocated.

So after all the JSON data is downloaded and parsed, we pass the list of jokes to the onDataUpdate property which then calls the render() method which gets the lists of jokes and reloads the table so the cells are populated with the data.

Although a bit more complex, it allows us to keep the more modular structure of having our data handler methods separate from the view controller.