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

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON
To display the JSON formatted in Chrome you'll need to install an extension.

Concurrency in Swift

Calling functions synchronously blocks the thread so you can't do anything else until the function finishes. It's really important to only use the main queue for the UI, otherwise other tasks may make the app unresponsive and slow as it's waiting for the other tasks to complete.
Asynchronous functions can be suspended while they're executing so they allow other work to continue in the thread so the UI won't freeze and be unresponsive. So you should use asynchronous functions if they might not be instantaneous, such as any networking calls.

Previous versions of Swift used completion handlers to handle asynchronous calls but Swift 5.5 added concurrency features including async/await so it's even easier than before to make asynchronous calls and write code that runs asynchronously.
The **async** keyword marks a method as asynchronous.
The **await** keyword must be used when calling an asynchronous method marked async to ensure the program waits for the async method to be completed before continuing. (similar to the try/throws relationship).
This means that the current execution will be paused until the async method completes.
Task is a unit of asynchronous work and are useful to call asynchronous methods from a synchronous context.

Swift async/await rules:

- Deployment target must be iOS 15.0 or later
- Along with methods, properties and initializers can be async as well
- Can't call asynchronous functions from a context that's not asynchronous
  - Create an asynchronous Task object to call asynchronous functions from a synchronous context. The Task object will automatically run everything in it asynchronously.
- Async functions can throw errors as well
  - Mark asynchronous functions that might throw errors as **async throws**
  - Call asynchronous functions that might throw errors with **try await**

- A do/catch statement is still a good structure for catching possible thrown errors

Actors are reference types and work similarly to classes but are safe to use in concurrent environments. Unlike classes, actors will ensure that only 1 task can mutate the actors' state at a time. This eliminates multiple tasks accessing/changing the same object state at the same time.
You might have seen the @MainActor keyword in the SDK docs. MainActor is a special kind of actor that always runs on the main thread. So all UIKit components are marked as MainActor since we always want the UI to be responsive.

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

1. URL and URLRequest encapsulates information about a URL request. URLRequest includes the ability to include the HTTP method and HTTP headers.
   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 provides two methods that downloads the data at a URL asynchronously
     - URLSession.shared.data(from: delegate:)
       - Takes a URL object
     - URLSession.shared.data(for: delegate:)
       - Takes a URLRequest object
     - Returns a tuple if successful, otherwise an error
       - Data – holds the downloaded data if successful
       - URLResponse - 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
       - Error – an error object if the request fails, data will be nil

In summary, making an HTTP request in iOS:
- create and configure an instance of URLSession, to make one or more HTTP requests
- create and configure an instance of URL or URLRequest for your requests. URLRequest is needed only if you need some specific parameters.
- Use the shared singleton session in URLSession to call one of the two methods that downloads the data asynchronously.
- If successful parse the JSON and update the app's UI accordingly.

JSON in iOS
Once the JSON has been downloaded successfully we need to parse the data and save it in an instance of our data model. 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
Language: Swift
Uncheck core data and include tests.
Uncheck create local git repo

Setup
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 jokeTableView.
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 "jokeCell".
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.
JokeAPI Documentation https://sv443.net/jokeapi/v2/
If you go to the "Try it out here" section you can see the options, the URL, and the result.
You can look at the results to see the structure.
An object is returned with key/value pairs.
The first key is "error", the second is "amount", and the third is "jokes"
The value for "jokes" is an array. Each item in the array is an object with key/value pairs. Keys:

- category
- type
- setup
- delivery
- flags

- id
- safe
- lang

Note that if the type is “single” the data won’t have setup and delivery, it will just have joke.
I want setup and delivery so I’m going to request the type “twopart”.
I only need the setup and the delivery so that’s what I’ll include in my data model struct.
Note that you must use the same structure and these keys exactly in your struct for the decoder to work or use decoding keys.

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 delivery: String
}
```

To use a JSONDecoder we need a data structure that mirrors the structure of the data we’re getting from the API. So we need a property named jokes and the value an array of our Joke struct.

```
struct JokeData: Decodable {
    var jokes = [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: "jokeCell",
for: indexPath)
        var cellConfig = cell.defaultContentConfiguration()
        cellConfig.text = jokes[indexPath.row].setup
        cell.contentConfiguration = cellConfig
        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?",
delivery: "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?", delivery: "Inheritance")
        let joke3 = Joke(setup: "If you put a million monkeys at a million
keyboards, one of them will eventually write a Java program.", delivery:
"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()
        loadtestdata()
    }
```

Run the app and you should see the jokes in the table. If there are any layout issues or table attributes you want to change you should do it now and work through these issue before using the API in case it has a limited number of 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].delivery, 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 delivery 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() async {
        //based on API documentation
        //https://sv443.net/jokeapi/v2/

        guard let urlPath = URL(string:
"https://v2.jokeapi.dev/joke/Programming?type=twopart&amount=10")
            else {
                print("url error")
                return
            }

        do {
            let (data, response) = try await URLSession.shared.data(from:
urlPath, delegate: nil)
            guard (response as? HTTPURLResponse)?.statusCode == 200
                else {
                    print("file download error")
                    return
                }
            //download successful
            print("download complete")
            parsejson(data)
        }
        catch {
            print(error.localizedDescription)
        }
    }
}
```

Because this method involves a network call that could take time we don't want it to run on the main thread and hold up the rest of the app so we use the "async" keyword to call it asynchronously. This will allow it to run on another thread and not hold up the main thread.
`URLSession.shared.data(from: delegate:)` is marked async so we need to call it with the "await" keyword. This method is also a throwing method so we need to call it with "try". The order must be "try await" in this case.
And we put this in a do/catch statement so we can handle the error if one is thrown.

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.jokes
            {
```

```
                jokeData.jokes.append(joke)
                print(joke)
            }
            print(jokeData.jokes.count)
            print("parsejson done")
        }
        catch let jsonErr
        {
            print(jsonErr.localizedDescription)
            return
        }
    }

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

We call parsejson() from loadjson() right after the print statement that the download was successful.

```
parsejson(data)
```

In ViewController.swift we need an instance of our JokeDataHandler class.

```
var jokeDataHandler = JokeDataHandler()
```

Now we want to update viewDidLoad() to use the API data instead of the test data. Comment out the call to loadtestdata() if you were using that.

Let's create a method to load the API data and call it from viewDidLoad().

```
    func getAPIdata() {
        Task {
            await jokeDataHandler.loadjson()
            jokes = jokeDataHandler.getJokes()
            print("Number of jokes \(jokes.count)")
        }
    }

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

Since loadjson() is marked to run asynchronously we need to call it with the "await" keyword. This also is helpful because we don't want to call getJokes() until the json has been loaded and parsed.
But if we add "await" we'll get an error because you can't call an asynchronous method from a method that's not asynchronous itself. If we tried to mark getAPIdata() async then we'll get that error in viewDidLoad() where we call getAPIdata(). Instead we wrap it in a Task{} object which creates a unit of asynchronous work so we can call asynchronous methods from a synchronous context.

If you run it now you'll probably see an empty table view. But if you add 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, the tableView datasource methods are still being called right away, so they're being called before the json is loaded, parsed, and the jokes array is populated. If you put in some print debugging statements you'll see when these are called. You can see that when tableView(_, numberOfRowsInSection) is called the jokes array has 0 items.
Why do you think tableView(_, cellForRowAt) is not called?
How can we fix this?
In getAPIdata() after we call getJokes() the jokes array will be populated so we can reload our table.

```
    func getAPIdata() {
        Task {
            await jokeDataHandler.loadjson()
            jokes = jokeDataHandler.getJokes()
            jokeTableView.reloadData()
            print("Number of jokes \(jokes.count)")
        }
    }
```

Reloading the table automatically calls the data source methods again. This time there are 10 items in the array so tableView(_, numberOfRowsInSection) returns 10 and tableView(_, cellForRowAt) will be called 10 times to draw each row.

More complex API

The API we used was about as simple an API as there is. Many APIs require API keys, and some have additional information that needs to be sent to the API so you need to read the documentation for each API to understand how to use it.

Documentation: https://rapidapi.com/KegenGuyll/api/dad-jokes/
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 "success" and the second is "body".
The value for "body" is an array. Each item in the array is an object with key/value pairs. Keys:

- _id
- setup
- punchline
- type

  and others …

I only need the setup and the punchline so that's what I'll include in my struct.
The differences in this example are:
jokes -> body
delivery -> punchline
I've made these changes throughout my project.
My data model now looks like this:

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

struct JokeData: Decodable {
```

```
    var body = [Joke]()
}
```

Through RapidAPI I needed to set up an account and get an API key for this API.
Now I can use their web site to test the API and they provide some code snippets.
Every API will have some documentation that you'll need to read in order to understand how to use the API and the structure of the data being sent back to you.
The Swift code snippet is using an older way to make asynchronous calls but it's still useful to see that we'll need to use a URLRequest object so we can set its headers which includes the API key. Some APIs will let you send the key in the URL but defining it in the header is more secure so many have moved to that method.
So our loadjson() method is updated to incorporate those changes.

```
    func loadjson() async {
        //based on API documentation
        //https://rapidapi.com/KegenGuyll/api/dad-jokes/

        let headers = [
            "x-rapidapi-key":
"09f9c252f0msh84e015c0d48e793p1e3311jsnb44f6cd4ae10",
            "x-rapidapi-host": "dad-jokes.p.rapidapi.com"
        ]

        let urlPath = "https://dad-jokes.p.rapidapi.com/random/joke?count=5"
        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

        do {
            let (data, response) = try await URLSession.shared.data(for:
request, delegate: nil)
            guard (response as? HTTPURLResponse)?.statusCode == 200
                else {
                    print("file download error")
                    return
                }
            //download successful
            print("download complete")
            parsejson(data)
        }
        catch {
            print(error.localizedDescription)
        }
    }
```