TP 2 - Client–Server communication

Adrien Humilière 22/02/2019

Part I. HTTP Requests

The objective of this lab is to use `URLSession` to pull data down from the iTunes API and display it in the console. You'll use a search query dictionary to configure a URL, which the URL session will use to fetch and print the correct data. Create a new playground called "iTunes Search," and set its indefinite execution property to `true`. This step will ensure that your playground will run continuously—so it can make network calls to an API at any time. Your code will look something like this:

```
import PlaygroundSupport

PlaygroundPage.current.needsIndefiniteExecution = true
```

Step 1: Review the iTunes Search API

- Take a few minutes to review the documentation for the **iTunes Search API**. Find the base URL for search requests, and pay particular attention to the types of queries you can make.
- Look at the different parameter keys listed in the documentation and their corresponding values. The different keys (term, media, limit, etc.) will be used when you create your query dictionary. Think of some parameters that you might want to include in your query.
- Create a query `[String: String]` dictionary that looks for your favorite movie or for songs by your favorite music artist. Make sure to use the exact keys and expected values listed in the API documentation. At a minimum, you'll want to use the `term` and `media` keys.
- Now that you have your query dictionary, create a variable to hold the base URL for the iTunes Search API: `https://itunes.apple.com/search?`.
- To make it easier to configure your URL properly, add an extension to the `URL` type with a function that returns a `URL?` based on a `[String: String]` query dictionary. You can use the same extension displayed in the slides of this lesson.

```swift
extension URL {
    func withQueries(_ queries: [String: String]) -> URL? {
        var components = URLComponents(url: self,
        resolvingAgainstBaseURL: true)
        components?.queryItems = queries.map
        { URLQueryItem(name: $0.0, value: $0.1) }
        return components?.url
    }
}
```

- With your base URL and query properties, create the search URL that you'll use to request data from the API.

Step 2: Pulling Data from the Web

- Now that you have your URL configured correctly, you'll use it to fetch your data from the web.
- Use the shared URLSession to create a dataTask for the specified URL. The completionHandler will give you three properties to work with: Data?, URLResponse?, and Error?. Provide appropriate names for the placeholder values.
- Don't forget to resume your dataTask.
- You're now ready to check if the dataTask has completed with valid data. Unwrap the data you received. If the data exists, create a string from the data that will display the data's contents, then print that string to the console.
- Once you've printed data to the console, your playground no longer needs to continue executing. You can stop the playground from running by adding the following line of code:

```swift
PlaygroundPage.current.finishExecution()
```

Part II.
Parsing JSON

The purpose of this lab is to get familiar with decoding JSON data into your own custom model objects. You'll start with the iTunes Search playground you created in the first part. Your task with this lab will be to decode the data you retrieved into a custom model object.

Step 1: Create Your Model Object

- Look through the data that you printed to the console in the previous lab. Identify what properties your object will have. Some of the properties might include title, artist, kind, description, and the URL for the artwork.
- Once you've identified the properties that will make up your object, create a `StoreItem` structure with those properties.
- You'll be making these `StoreItem` objects from the JSON data, so you'll add custom keys as an enumeration on your object.
- Now create your own implementation of `init(from:)` that takes a `Decoder` as a parameter. The implementation of this should get the `KeyedCodingContainer` from `decoder` and then use it to decode the values for each key. The values should then be assigned to their corresponding properties.
- You might have noticed that an item can have different types of `descriptions`. Some items have a description key, while others have a `longDescription` key. Your model object makes no distinction between the two. If the API sends down a value with the `description` key, that is what you should assign to your model. However, if it sends down a value with the `longDescription` key and nothing with the `description` key, you should use that instead. If neither value exists, just assign `description` to an empty string.
- To do this, you will need a `CodingKey` for each. But the `Codable` protocol doesn't allow you to add cases to `CodingKeys` that don't match one of your properties, i.e. `longDescription`. To get around this, create a second nested enumeration called `AdditionalKeys` that conforms to `CodingKey` and has a case for `longDescription`. Then, in `init(from:)` you can use `try?` when decoding the value for `description`, and if it fails you can decode the value associated with `longDescription` and assign it to the description property.
- There is one last hurdle to jump over. Each `KeyedCodingContainer` only contains the key/value pairs associated with the cases in the `CodingKey` enumeration used, so to decode the value for `longDescription`, you need to get a different `KeyedCodingContainer` from `decoder` using `AdditionalKeys`. When all of this is done, your `AdditionalKeys` enumeration and `init(from:)` initializer should look similar to the following:

```swift
enum AdditionalKeys: String, CodingKey {
    case longDescription
}

init(from decoder: Decoder) throws {
    let values = try decoder.container(keyedBy: CodingKeys.self)
    name = try values.decode(String.self, forKey:
       CodingKeys.name)
    artist = try values.decode(String.self, forKey:
       CodingKeys.artist)
    kind = try values.decode(String.self, forKey:
       CodingKeys.kind)
    artworkURL = try values.decode(URL.self, forKey:
       CodingKeys.artworkURL)

    if let description = try? values.decode(String.self, forKey:
       CodingKeys.description) {
        self.description = description
    } else {
        let additionalValues = try decoder.container(keyedBy:
           AdditionalKeys.self)
        description = (try? additionalValues.decode(String.self,
           forKey: AdditionalKeys.longDescription)) ?? ""
    }
}
```

Step 2: Create a Function to Fetch Items

Now that you have a `StoreItem` object, you're ready to fetch the data that you'll decode into your model type.

- Create a new `fetchItems` function that takes a query dictionary and a completion handler. The completion handler should accept an optional array of store items (`[StoreItem]`). The completion handler should be marked as `@escaping`.

```
func fetchItems(matching query: [String: String], completion:
@escaping ([StoreItem]?) -> Void) { }
```

The function should create a URL with the queries, create a new request, and resume the request. When the request is complete, the function should create a `JSONDecoder`, unwrap the data, and decode it into an array of `StoreItem` objects.

The response from the iTunes Search API is structured a little differently from the NASA APOD API. The first difference is that the results are under the `"results"` key, rather than at the top level. The second difference is that the results come back as an array, not as an individual object.

To accommodate this, you'll need to create an intermediary object that conforms to `Codable` that uses `results` as a key and has a value of type `[StoreItem]`. The easiest way to do this is to create a simple struct called StoreItems that has only a property results of type [StoreItem]. Since StoreItem conforms to Codable and the property name results is the same as the key in the data, you simply need to put Codable in your StoreItems declaration and the compiler will auto-generate the rest of what is needed at compile-time for a Decoder object to be able to decode data to a StoreItems object. This will look as follows:

```swift
struct StoreItems: Codable {
    let results: [StoreItem]
}
```

Follow these steps to implement the fetchItems function:

- If you haven't already, create a StoreItems struct as shown previously.
- Move your baseURL variable inside the function.
- Instantiate a new URL with the queries that were passed in using the withQueries function. You'll want to guard against any URLs that can't be constructed with the supplied queries.
- Create a new data task on the shared URL session. (You'll implement the completion handler in the next step.)
- Resume the task.

Step 3: Decode the JSON

- Inside the task's closure, create a new JSONDecoder.
- Check to see if you've received valid data back from the API. If you have, try and decode it into a StoreItems object.
- If you successfully unwrapped both the data and the StoreItems object, pass the results property on StoreItems through the completion handler. Remember that the results property on StoreItems has the array of each individual StoreItem object.
- If the data can't be unwrapped or if the JSON fails to decode properly, print an explanation to the console (something like the response was invalid or could not be decoded), and call the completion handler with a nil parameter.

Step 4: Call Your Function

With the StoreItem struct, the intermediary StoreItems struct, the fetchItems function, and your search query, you're ready to call your function and create your model objects.

- Call the fetchItems function and pass in your query. Print the decoded [StoreItem] array in the completion handler.

You did it! You've fetched data and decoded it into your own custom model object. You'll use code just like this for fetching data to display in your apps.