Sep 12, 2016

Working with JSON in Swift

If your app communicates with a web application, information returned from the server is often formatted as JSON. You can use the Foundation framework's JSONSerialization class to convert JSON into Swift data types like `Dictionary`, `Array`, `String`, `Number`, and `Bool`. However, because you can't be sure of the structure or values of JSON your app receives, it can be challenging to deserialize model objects correctly. This post describes a few approaches you can take when working with JSON in your apps.

Extracting Values from JSON

The `JSONSerialization` class method `jsonObject(with:options:)` returns a value of type `Any` and throws an error if the data couldn't be parsed.

```
import Foundation

let data: Data // received from a network request, for example
let json = try? JSONSerialization.jsonObject(with: data, options: [])
```

Although valid JSON may contain only a single value, a response from a web application typically encodes an object or array as the top-level object. You can use optional binding and the `as?` type cast operator in an `if` or `guard` statement to extract a value of known type as a constant. To get a `Dictionary` value from a JSON object type, conditionally cast it as `[String: Any]`. To get an `Array` value from a JSON array type, conditionally cast it as `[Any]` (or an array with a more specific element type, like `[String]`). You can extract a dictionary value by key or an array value by index using type cast optional binding with subscript accessors or pattern matching with enumeration.

```
// Example JSON with object root:
/*
    {
        "someKey": 42.0,
        "anotherKey": {
            "someNestedKey": true
        }
    }
*/
if let dictionary = jsonWithObjectRoot as? [String: Any] {
    if let number = dictionary["someKey"] as? Double {
        // access individual value in dictionary
    }

    for (key, value) in dictionary {
```

```
        // access all key / value pairs in dictionary
    }

    if let nestedDictionary = dictionary["anotherKey"] as? [String: Any] {
        // access nested dictionary values by key
    }
}

// Example JSON with array root:
/*
    [
        "hello", 3, true
    ]
*/
if let array = jsonWithArrayRoot as? [Any] {
    if let firstObject = array.first {
        // access individual object in array
    }

    for object in array {
        // access all objects in array
    }

    for case let string as String in array {
        // access only string values in array
    }
}
```

Swift's built-in language features make it easy to safely extract and work with JSON data decoded with Foundation APIs — without the need for an external library or framework.

Creating Model Objects from Values Extracted from JSON

Since most Swift apps follow the Model-View-Controller design pattern, it is often useful to convert JSON data to objects that are specific to your app's domain in a model definition.

For example, when writing an app that provides search results for local restaurants, you might implement a `Restaurant` model with an initializer that accepts a JSON object and a type method that makes an HTTP request to a server's `/search` endpoint and then asynchronously returns an array of `Restaurant` objects.

Consider the following `Restaurant` model:

```
import Foundation

struct Restaurant {
    enum Meal: String {
        case breakfast, lunch, dinner
    }

    let name: String
    let location: (latitude: Double, longitude: Double)
```

```
    let meals: Set<Meal>
}
```

A `Restaurant` has a `name` of type `String`, a `location` expressed as a coordinate pair, and a `Set` of `meals` containing values of a nested `Meal` enumeration.

Here's an example of how a single restaurant may be represented in a server response:

```
{
    "name": "Caffè Macs",
    "coordinates": {
        "lat": 37.330576,
        "lng": -122.029739
    },
    "meals": ["breakfast", "lunch", "dinner"]
}
```

Writing an Optional JSON Initializer

To convert from a JSON representation to a `Restaurant` object, write an initializer that takes an `Any` argument that extracts and transforms data from the JSON representation into properties.

```
extension Restaurant {
    init?(json: [String: Any]) {
        guard let name = json["name"] as? String,
            let coordinatesJSON = json["coordinates"] as? [String: Double],
            let latitude = coordinatesJSON["lat"],
            let longitude = coordinatesJSON["lng"],
            let mealsJSON = json["meals"] as? [String]
        else {
            return nil
        }

        var meals: Set<Meal> = []
        for string in mealsJSON {
            guard let meal = Meal(rawValue: string) else {
                return nil
            }

            meals.insert(meal)
        }

        self.name = name
        self.coordinates = (latitude, longitude)
        self.meals = meals
    }
}
```

If your app communicates with one or more web services that do not return a single, consistent representation of a model object, consider implementing several initializers to handle each of the possible representations.

In the example above, each of the values are extracted into constants from the passed JSON dictionary using optional binding and the `as?` type casting operator. For the `name` property, the extracted `name` value is simply assigned as-is. For the `coordinate` property, the extracted `latitude` and `longitude` values are combined into a tuple before assignment. For the `meals` property, the extracted string values are iterated over to construct a `Set` of `Meal` enumeration values.

Writing a JSON Initializer with Error Handling

The previous example implements an optional initializer that returns `nil` if deserialization fails. Alternatively, you can define a type conforming to the `Error` protocol and implement an initializer that throws an error of that type whenever deserialization fails.

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

extension Restaurant {
    init(json: [String: Any]) throws {
        // Extract name
        guard let name = json["name"] as? String else {
            throw SerializationError.missing("name")
        }

        // Extract and validate coordinates
        guard let coordinatesJSON = json["coordinates"] as? [String: Double],
            let latitude = coordinatesJSON["lat"],
            let longitude = coordinatesJSON["lng"]
        else {
            throw SerializationError.missing("coordinates")
        }

        let coordinates = (latitude, longitude)
        guard case (-90...90, -180...180) = coordinates else {
            throw SerializationError.invalid("coordinates", coordinates)
        }

        // Extract and validate meals
        guard let mealsJSON = json["meals"] as? [String] else {
            throw SerializationError.missing("meals")
        }

        var meals: Set<Meal> = []
        for string in mealsJSON {
            guard let meal = Meal(rawValue: string) else {
                throw SerializationError.invalid("meals", string)
            }

            meals.insert(meal)
        }

        // Initialize properties
```

```
        self.name = name
        self.coordinates = coordinates
        self.meals = meals
    }
}
```

Here, the `Restaurant` type declares a nested `SerializationError` type, which defines enumeration cases with associated values for missing or invalid properties. In the throwing version of the JSON initializers, rather than indicating failure by returning `nil`, an error is thrown to communicate the specific failure. This version also performs validation of input data to ensure that `coordinates` represents a valid geographic coordinate pair and that each of the names for `meals` specified in the JSON correspond to `Meal` enumeration cases.

Writing a Type Method for Fetching Results

A web application endpoint often returns multiple resources in a single JSON response. For example, a `/search` endpoint may return zero or more restaurants that match the requested query parameter and include those representations along with other metadata:

```
{
    "query": "sandwich",
    "results_count": 12,
    "page": 1,
    "results": [
        {
            "name": "Caffè Macs",
            "coordinates": {
                "lat": 37.330576,
                "lng": -122.029739
            },
            "meals": ["breakfast", "lunch", "dinner"]
        },
        ...
    ]
}
```

You can create a type method on the `Restaurant` structure that translates a `query` method parameter into a corresponding request object and sends the HTTP request to the web service. This code would also be responsible for handling the response, deserializing the JSON data, creating `Restaurant` objects from each of the extracted dictionaries in the `"results"` array, and asynchronously returning them in a completion handler.

```
extension Restaurant {
    private let urlComponents: URLComponents // base URL components of the web service
    private let session: URLSession // shared session for interacting with the web service

    static func restaurants(matching query: String, completion: ([Restaurant]) -> Void) {
        var searchURLComponents = urlComponents
        searchURLComponents.path = "/search"
        searchURLComponents.queryItems = [URLQueryItem(name: "q", value: query)]
        let searchURL = searchURLComponents.url!
```

```
        session.dataTask(url: searchURL, completion: { (_, _, data, _)
            var restaurants: [Restaurant] = []

            if let data = data,
                let json = try? JSONSerialization.jsonObject(with: data, options: []) as? [String: Any] {
                for case let result in json["results"] {
                    if let restaurant = Restaurant(json: result) {
                        restaurants.append(restaurant)
                    }
                }
            }

            completion(restaurants)
        }).resume()
    }
}
```

A view controller can call this method when the user enters text into a search bar to populate a table view with matching restaurants:

```
import UIKit

extension ViewController: UISearchResultsUpdating {
    func updateSearchResultsForSearchController(_ searchController: UISearchController) {
        if let query = searchController.searchBar.text, !query.isEmpty {
            Restaurant.restaurants(matching: query) { restaurants in
                self.restaurants = restaurants
                self.tableView.reloadData()
            }
        }
    }
}
```

Separating concerns in this way provides a consistent interface for accessing restaurant resources from view controllers, even when the implementation details about the web service change.

Reflecting on Reflection

Converting between representations of the same data in order to communicate between different systems is a tedious, albeit necessary, task for writing software.

Because the structure of these representations can be quite similar, it may be tempting to create a higher-level abstraction to automatically map between these different representations. For instance, a type might define a mapping between `snake_case` JSON keys and `camelCase` property names in order to automatically initialize a model from JSON using the Swift reflection APIs, such as `Mirror`.

However, we've found that these kinds of abstractions tend not to offer significant benefits over conventional usage of Swift language features, and instead make it more difficult to debug problems or handle edge cases. In the example above, the initializer not only extracts and maps values from JSON, but also initializes complex data types and performs domain-specific input validation. A reflection-based approach would have to go to great lengths in

order to accomplish all of these tasks. Keep this in mind when evaluating the available strategies for your own app. The cost of small amounts of duplication may be significantly less than picking the incorrect abstraction.

All Blog Posts