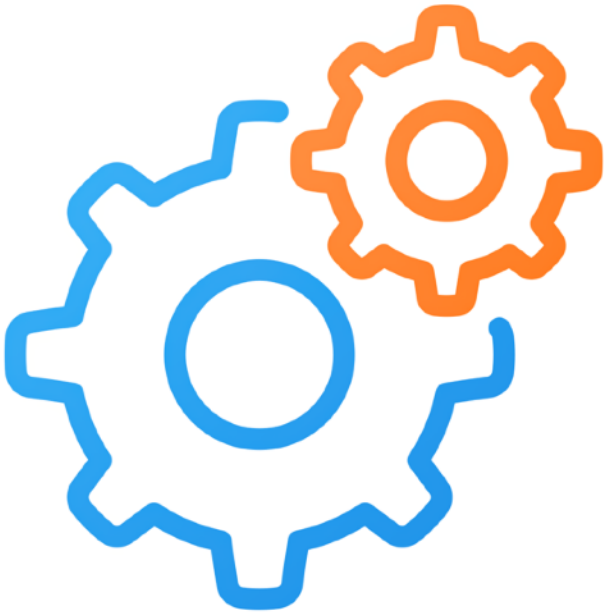
Matteo Manferdini



Parsing JSON in Swift

The Ultimate Cheat Sheet

A Small Promise

It took me weeks of research and many hours of work to create this guide. But I am happy to offer it to you completely free of charge.

In exchange, I ask you for a small promise.

If you think I did a good job, please share this with someone who can benefit using this link:

https://matteomanferdini.com/parsing-json-in-swift-cheat-sheet/

That way, I can spend less time looking for people I can help and more producing great free material.

Thank you for reading.

Let's start.

Basic JSON Parsing in Swift

JSON parsing with Decodable types

First, declare a Swift type (structure or class) that conforms to the `Decodable` protocol and has stored properties corresponding to the members in the JSON object.

Then, decode the data using an instance of the `JSONDecoder` class.

```
import Foundation

let json = """
{
    "objectNumber": "SK-A-1718",
    "title": "Winter Landscape with Ice Skaters",
    "plaqueDescriptionEnglish": "Hendrick Avercamp turned the winter
        landscape into a subject in its own right.",
    "principalMaker": "Hendrick Avercamp"
}
""".data(using: .utf8)!

struct Painting: Decodable {
    let objectNumber: String
    let title: String
    let plaqueDescriptionEnglish: String
    let principalMaker: String
}

let painting = try JSONDecoder().decode(Painting.self, from: json)
```

Further reading: Parsing JSON in Swift: The Complete Guide [With Examples]

Renaming snake case members to camel case

Set the `keyDecodingStrategy` property of `JSONDecoder` to `.convertFromSnakeCase`

```
import Foundation

let json = """
{
    "first_name": "Rembrandt",
    "last_name": "van Rijn",
    "profession": "Painter"
}
""".data(using: .utf8)!

struct Artist: Decodable {
    let firstName: String
    let lastName: String
    let profession: String
}

let decoder = JSONDecoder()
decoder.keyDecodingStrategy = .convertFromSnakeCase
let artist = try decoder.decode(Artist.self, from: json)
```

Parsing JSON without Decodable types

The `JSONSerialization` class does not require custom Swift types to decode JSON data. Instead, it converts it into dictionaries, arrays, and other standard Swift types.

```
import Foundation

let data = """
{
    "name": "Tomatoes",
    "nutrition": {
        "calories": 22,
        "carbohydrates": 4.8,
        "protein": 1.1,
        "fat": 0.2
    },
    "tags": [
        "Fruit",
        "Vegetable",
        "Vegan",
        "Gluten free"
    ]
}
""".data(using: .utf8)!

let json = try JSONSerialization.jsonObject(with: data)
    as! [String: Any]
let nutrition = json["nutrition"] as! [String: Any]
let carbohydrates = nutrition["carbohydrates"] as! Double
print(carbohydrates)
// Output: 4.8
```

Further reading: Parse JSON in Swift without Codable [Arrays and Dictionaries]

Optional keys and null values

In your `Decodable` Swift type, you only need stored properties for the JSON object members you want to parse. Any other field will be ignored.

If a member of a JSON object is `null` or missing, you can make the corresponding property of your Swift type optional.

```
import Foundation

let json = """
{
    "name": "Hendrick Avercamp",
    "placeOfBirth": null,
}
""".data(using: .utf8)!

struct Maker: Decodable {
    let name: String
    let placeOfBirth: String?
    let placeOfDeath: String?
}

let maker = try JSONDecoder().decode(Maker.self, from: json)
```

Decoding nested JSON objects using nested Swift types

Create Swift types that match the JSON data's nesting structure.

```
import Foundation

let json = """
{
    "count": 2,
    "paintings": [
        {
            "objectNumber": "SK-A-1718",
            "title": "Winter Landscape with Ice Skaters",
            "maker": "Hendrick Avercamp"
        },
        {
            "objectNumber": "SK-C-109",
            "title": "Italian Landscape with a Draughtsman",
            "maker": "Jan Both"
        }
    ]
}
""".data(using: .utf8)!

struct Result: Decodable {
    let count: Int
    let paintings: [Painting]
}

struct Painting: Decodable {
    let objectNumber: String
    let title: String
    let maker: String
}

let maker = try JSONDecoder().decode(Result.self, from: json)
```

Customizing the Decoding with Coding Keys

Renaming the fields of a JSON object

Name your coding keys following your type's property names and use a `String` raw value for the corresponding field in the JSON data.

```swift
import Foundation

let json = """
{
    "first_name": "Rembrandt",
    "last_name": "van Rijn",
    "profession": "Painter"
}
""".data(using: .utf8)!

struct Artist: Decodable {
    let name: String
    let surname: String
    let profession: String

    enum CodingKeys: String, CodingKey {
        case name = "first_name"
        case surname = "last_name"
        case profession
    }
}

let artist = try JSONDecoder().decode(Artist.self, from: json)
```

Selecting which stored properties to decode

The `JSONDecoder` class tries to decode all the properties of a `Decodable` type. If any of those properties are absent in the JSON data, it will throw an error.

Use coding keys to indicate which properties to decode.

```
import Foundation

let json = """
{
    "name": "Rembrandt Harmenszoon van Rijn",
    "profession": "Painter"
}
""".data(using: .utf8)!

struct Artist: Decodable {
    let name: String
    let profession: String
    var viewCount: Int = 0
    var isFavorite: Bool = false

    enum CodingKeys: CodingKey {
        case name
        case profession
    }
}

let artist = try JSONDecoder().decode(Artist.self, from: json)
```

Adding custom logic with a decoding initializer

Provide coding keys matching the fields in the JSON data. Then, add a decoding initializer where you perform additional computations.

```
import Foundation

let json = """
{
    "first_name": "Rembrandt",
    "middle_name": "Harmenszoon",
    "last_name": "van Rijn",
}
""".data(using: .utf8)!

struct Artist: Decodable {
    let name: String
    let profession: String?

    enum CodingKeys: String, CodingKey {
        case firstName
        case middleName
        case lastName
        case profession
    }

    init(from decoder: any Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        let firstName = try container.decode(String.self,
            forKey: .firstName)
        let middleName = try container.decode(String.self,
            forKey: .middleName)
        let lastName = try container.decode(String.self,
            forKey: .lastName)
        self.name = firstName + " " + middleName + " " + lastName
        self.profession = try container.decodeIfPresent(String.self,
            forKey: .profession)
    }
}
```

```
let decoder = JSONDecoder()
decoder.keyDecodingStrategy = .convertFromSnakeCase
let artist = try decoder.decode(Artist.self, from: json)
```

Further reading: Coding Keys in Swift with Decodable: How and When to Use Them

Working with Date Formats

ISO 8601 format (without fractional seconds)

Dates in the ISO 8601 format usually have the form `yyyy-MM-ddTHH:mm:ssZ`

```
import Foundation

let jsonData = """
{
 "date": "2024-10-26T12:00:00Z"
}
""".data(using: .utf8)!

struct MyStruct: Codable {
    let date: Date
}

let decoder = JSONDecoder()
decoder.dateDecodingStrategy = .iso8601
let decodedData = try decoder.decode(MyStruct.self, from: jsonData)
print(decodedData.date)
// Output: 2024-10-26 12:00:00 +0000
```

ISO 8601 formats not supported by the .iso8601 decoding strategy

Use the `.custom(_:)` date decoding strategy with an instance of the `ISO8601DateFormatter` class.

For example, ISO 8601 dates with fractional seconds.

```
import Foundation

let jsonData = """
{
 "date": "2024-10-26T12:34:56.789Z"
}
""".data(using: .utf8)!

struct MyStruct: Codable {
    let date: Date
}

let decoder = JSONDecoder()
decoder.dateDecodingStrategy = .custom({ decoder in
    let container = try decoder.singleValueContainer()
    let dateString = try container.decode(String.self)
    let formatter = ISO8601DateFormatter()
    formatter.formatOptions = [.withInternetDateTime,
        .withFractionalSeconds]
    return formatter.date(from: dateString)!
})
let decodedData = try decoder.decode(MyStruct.self, from: jsonData)
print(decodedData.date)
// Output: 2024-10-26 12:34:56 +0000
```

UNIX dates expressed in seconds or milliseconds

Since most internet servers are UNIX-based, they use UNIX time, which is the number of seconds that passed since midnight on January 1st, 1970 (UTC).

The `JSONDecoder` class offers two granularities for UNIX time: `.secondsSince1970` and `.millisecondsSince1970`.

```
import Foundation

let jsonData = """
{
 "date": 1729936800
}
""".data(using: .utf8)!

struct MyStruct: Codable {
    let date: Date
}

let decoder = JSONDecoder()
decoder.dateDecodingStrategy = .secondsSince1970
let decodedData = try decoder.decode(MyStruct.self, from: jsonData)
print(decodedData.date)
// Output: 2024-10-26 12:00:00 +0000
```

Custom string dates

The `.formatted(_:)` date decoding strategy accepts a `DateFormatter` instance.

```
import Foundation

let jsonData = """
{
 "date": "2024-10-26 12:00:00"
}
""".data(using: .utf8)!

struct MyStruct: Codable {
    let date: Date
}

let formatter = DateFormatter()
formatter.dateFormat = "yyyy-MM-dd HH:mm:ss"
let decoder = JSONDecoder()
decoder.dateDecodingStrategy = .formatted(formatter)
let decodedData = try decoder.decode(MyStruct.self, from: jsonData)
print(decodedData.date)
// Output: 2024-10-26 12:00:00 +0000
```

Mixed date formats

Use the `.custom(_:)` date decoding strategy to handle the various data formats.

If dates are all represented using the same data type, e.g., `String`, first decode the date value and then try multiple date formatters in sequence until the conversion succeeds.

If dates are represented by different data types, e.g., `String` and `TimeInterval`, check the coding key before you use the appropriate Swift type for the decoding.

```
import Foundation

let jsonData = """
{
 "ISODate": "2024-10-26T12:00:00Z",
 "UNIXDate": 1729936800
}
""".data(using: .utf8)!

struct MyStruct: Codable {
    let ISODate: Date
    let UNIXDate: Date

    enum CodingKeys: CodingKey {
        case ISODate
        case UNIXDate
    }
}

let decoder = JSONDecoder()
decoder.dateDecodingStrategy = .custom({ decoder in
    let container = try decoder.singleValueContainer()
    let codingKey = decoder.codingPath.last!
```

```
    if codingKey.stringValue == MyStruct.CodingKeys.ISODate.stringValue {
        let dateString = try container.decode(String.self)
        let formatter = ISO8601DateFormatter()
        formatter.formatOptions = [.withInternetDateTime]
        return formatter.date(from: dateString)!
    } else {
        let UNIXTime = try container.decode(TimeInterval.self)
        return Date(timeIntervalSince1970: UNIXTime)
    }
})

let decodedData = try decoder.decode(MyStruct.self, from: jsonData)
print(decodedData.ISODate)
// Output: 2024-10-26 12:00:00 +0000
print(decodedData.UNIXDate)
// Output: 2024-10-26 12:00:00 +0000
```

Further reading: Date decoding strategies in Swift [with Examples]

Advanced Decoding Techniques Using Containers

Flattening a single JSON object

In a decoding initializer, individually decode the properties of a nested JSON object using a `KeyedDecodingContainer`. You get this container by calling the `nestedContainer(keyedBy:forKey:)` method on the main container.

```
import Foundation

let json = """
{
    "title": "Inception",
    "year": 2010,
    "director": {
        "name": "Christopher",
        "surname": "Nolan"
    }
}
""".data(using: .utf8)!

struct Movie: Decodable {
    let title: String
    let year: Int
    let director: String

    enum CodingKeys: CodingKey {
        case title
        case year
        case director
    }
```

```
    enum PersonKeys: CodingKey {
        case name
        case surname
    }

    init(from decoder: any Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        title = try container.decode(String.self, forKey: .title)
        year = try container.decode(Int.self, forKey: .year)
        let directorContainer = try container.nestedContainer(
            keyedBy: PersonKeys.self,
            forKey: .director
        )
        let name = try directorContainer.decode(String.self,
            forKey: .name)
        let surname = try directorContainer.decode(String.self,
            forKey: .surname)
        self.director = name + " " + surname
    }
}

let movie = try JSONDecoder().decode(Movie.self, from: json)
```

Flattening an array of JSON objects

Call the `nestedUnkeyedContainer(forKey:)` method on the main container to get an `UnkeyedDecodingContainer` for a nested array of objects.

```
import Foundation

let json = """
{
    "title": "Inception",
    "year": 2010,
    "cast": [
        {
            "name": "Leonardo",
            "surname": "DiCaprio"
        },
        {
            "name": "Cillian",
            "surname": "Murphy"
        },
        {
            "name": "Michael",
            "surname": "Caine"
        }
    ]
}
""".data(using: .utf8)!

struct Person: Decodable {
    let name: String
    let surname: String
}

struct Movie: Decodable {
    let title: String
    let year: Int
    let cast: [String]
```

```
    enum CodingKeys: CodingKey {
        case title
        case year
        case cast
    }

    enum PersonKeys: CodingKey {
        case name
        case surname
    }

    init(from decoder: any Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        title = try container.decode(String.self, forKey: .title)
        year = try container.decode(Int.self, forKey: .year)

        var castContainer = try container.nestedUnkeyedContainer(
            forKey: .cast)
        var cast: [String] = []
        while !castContainer.isAtEnd {
            let actorContainer = try castContainer.nestedContainer(
                keyedBy: PersonKeys.self)
            let name = try actorContainer.decode(String.self,
                forKey: .name)
            let surname = try actorContainer.decode(String.self,
                forKey: .surname)
            cast.append(name + " " + surname)
        }
        self.cast = cast
    }
}

let movie = try JSONDecoder().decode(Movie.self, from: json)
```

Further reading: Decoding and Flattening Nested JSON with Codable

Decoding dynamic keys as a Swift dictionary

The nested JSON objects require a `Decodable` Swift type.

The parent object can instead be decoded as a dictionary with `String` or `Int` keys, depending on the kind of keys the JSON data uses.

You can then loop through the dictionary and transform its content into an array.

```
import Foundation

let json = """
{
    "Amsterdam": {
        "conditions": "Partly Cloudy",
        "temperature": 11
    },
    "New York": {
        "conditions": "Sunny",
        "temperature": 15,
    },
    "Moscow": {
        "conditions": "Cloudy",
        "temperature": 6,
    },
}
""".data(using: .utf8)!

struct WeatherReport: Decodable {
    let conditions: String
    let temperature: Int
}

let reports = try JSONDecoder().decode([String: WeatherReport].self,
    from: json)
```

Decoding dynamic keys as a Swift array

The Swift type for the nested JSON objects needs a custom decoding initializer that takes its identifier from the coding path provided by the `KeyedDecodingContainer`.

Then, create a `CodingKey` structure that can be initialized with a dynamic value. When decoding the main JSON object, use the dynamic `CodingKey` structure instead of a static enumeration.

Then, iterate through the container's `allKeys` property and use the keys to decode each nested JSON object.

```
import Foundation

let json = """
{
    "Amsterdam": {
        "conditions": "Partly Cloudy",
        "temperature": 11
    },
    "New York": {
        "conditions": "Sunny",
        "temperature": 15,
    },
    "Moscow": {
        "conditions": "Cloudy",
        "temperature": 6,
    },
}
""".data(using: .utf8)!

struct WeatherReport: Decodable {
    let city: String
    let conditions: String
    let temperature: Int
```

```
    enum CodingKeys: CodingKey {
        case conditions
        case temperature
    }

    init(from decoder: any Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        self.city = container.codingPath.first!.stringValue
        self.conditions = try container.decode(String.self,
            forKey: .conditions)
        self.temperature = try container.decode(Int.self,
            forKey: .temperature)
    }
}

struct WeatherInfo: Decodable {
    let reports: [WeatherReport]

    struct CodingKeys: CodingKey {
        var stringValue: String
        var intValue: Int? = nil

        init?(stringValue: String) {
            self.stringValue = stringValue
        }

        init?(intValue: Int) {
            return nil
        }
    }

    init(from decoder: any Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        var reports: [WeatherReport] = []
```

```
        for key in container.allKeys {
            let report = try container.decode(WeatherReport.self,
                forKey: key)
            reports.append(report)
        }
        self.reports = reports
    }
}

let reports = try JSONDecoder().decode(WeatherInfo.self, from: json)
```

Further reading: Decode JSON with Dynamic Keys in Swift [Dictionaries and Arrays]

Decoding JSON objects with dynamic types

Homogenize the dynamic types by making them descend from a common superclass or conform to the same protocols.

Then, determine the dynamic type of the nested objects and decode them in a decoding initializer in the Swift type for the parent JSON object.

```
import Foundation

let json = """
{
    "name": "John Doe",
    "posts": [
        {
            "type": "text",
            "value": {
                "content": "Hello, World!"
            }
        },
        {
            "type": "image",
            "value": {
                "url": "http://example.com/images/photo.jpeg"
            }
        }
    ]
}
""".data(using: .utf8)!

protocol Post: Decodable {}

struct Text: Post {
    let content: String
}
```

```
struct Image: Post {
    let url: URL
}

struct User: Decodable {
    let name: String
    let posts: [Post]

    enum CodingKeys: CodingKey {
        case name, posts
    }

    enum PostCodingKeys: CodingKey {
        case type, value
    }

    init(from decoder: any Decoder) throws {
        let container = try decoder.container(keyedBy: CodingKeys.self)
        self.name = try container.decode(String.self, forKey: .name)
        var postsContainer = try container.nestedUnkeyedContainer(
            forKey: .posts)
        var posts: [Post] = []
        while !postsContainer.isAtEnd {
            let postContainer = try postsContainer.nestedContainer(
                keyedBy: PostCodingKeys.self)
            let type = try postContainer.decode(String.self,
                forKey: .type)
            let post: Post = switch type {
            case "text":
                try postContainer.decode(Text.self, forKey: .value)
            case "image":
                try postContainer.decode(Image.self, forKey: .value)
```

```
                default:
                    throw DecodingError.dataCorruptedError(
                        forKey: .value,
                        in: postContainer,
                        debugDescription: "Unknown post type"
                    )
                }
                posts.append(post)
            }
            self.posts = posts
        }
}

let user = try JSONDecoder().decode(User.self, from: json)
```

Further reading: Decoding JSON with Dynamic Types in Swift [With Codable]

App Development Tasks

Formatting and pretty printing for debugging and testing

With Codable and JSONDecoder

```
import Foundation

struct Person: Codable {
    let name: String
    let age: Int
    let employers: [String]
}

let json = #"{"name":"John Doe","age":30,"employers":
["Apple","Facebook","Google"]}"#
let data = json.data(using: .utf8)!
let person = try JSONDecoder().decode(Person.self, from: data)
let encoder = JSONEncoder()
encoder.outputFormatting = [.prettyPrinted, .sortedKeys]
let prettyPrintedData = try encoder.encode(person)
let prettyPrintedString = String(data: prettyPrintedData,
encoding: .utf8)!
print(prettyPrintedString)
```

With JSONSerialization

```
import Foundation

let json = #"{"name":"John Doe","age":30,"employers":
["Apple","Facebook","Google"]}"#
let data = json.data(using: .utf8)!
let object = try JSONSerialization.jsonObject(with: data)
let prettyPrintedData = try JSONSerialization.data(
    withJSONObject: object,
    options: [.prettyPrinted, .sortedKeys]
)
let prettyPrintedString = String(data: prettyPrintedData,
encoding: .utf8)!
print(prettyPrintedString)
```

Further reading: Swift Data to JSON String: Formatting and Pretty Printing

Reading and writing JSON files

Generic functions

```
import Foundation

func readJSONFile<T: Decodable>(with url: URL) throws -> T {
    let data = try Data(contentsOf: url)
    return try JSONDecoder().decode(T.self, from: data)
}

func write<T: Encodable>(value: T, at url: URL) throws {
    let data = try JSONEncoder().encode(value)
    try data.write(to: url)
}
```

Reading a JSON file from the app's main bundle

```
struct User: Decodable {
    // ...
}

func readUserFromBundle() -> User? {
    guard let url = Bundle.main.url(forResource: "User",
        withExtension: "json") else {
        return nil
    }
    return try? readJSONFile(with: url)
}
```

Reading and writing a JSON file in the app's document directory

```
struct User: Decodable {
    // ...
}

extension User {
    static let documentsURL = URL.documentsDirectory
        .appendingPathComponent("User")
        .appendingPathExtension("json")
}

func writeUserIntoDocuments(user: User) {
    try? write(value: user, at: User.documentsURL)
}

func readUserFromDocuments() -> User? {
    return try? readJSONFile(with: User.documentsURL)
}
```

Further reading: Reading a JSON File in Swift [Bundle and Documents Directory]

Fetching and parsing data coming from a URL in SwiftUI

```
import Foundation
import SwiftUI

struct Response: Decodable {
    let results: [Book]
}

struct Book: Decodable, Identifiable {
    let id: Int
    let title: String
    let downloads: Int

    enum CodingKeys: String, CodingKey {
        case id, title
        case downloads = "download_count"
    }
}

struct ContentView: View {
    @State var books: [Book] = []

    var body: some View {
        List(books) { book in
            LabeledContent(book.title, value: book.downloads,
                format: .number)
                .monospacedDigit()
        }
        .task {
            books = (try? await fetchBooks()) ?? []
        }
    }
```

```
    func fetchBooks() async throws -> [Book] {
        let url = URL(string: "https://gutendex.com/books/")!
        let (data, _) = try await URLSession.shared.data(from: url)
        let response = try JSONDecoder().decode(Response.self, from: data)
        return response.results
    }
}
```

Further reading: Parse JSON from an API URL in Swift [Codable and URLSession]

Please Share this Guide

I hope you enjoyed this cheat sheet and that it helped you improve your understanding of decoding JSON data in Swift. It took me several hours to put it together, but I am happy to offer it to you free of charge.

If you found it useful, please take a moment to share it with someone who might also find it useful. This way, I can dedicate more time to creating more free articles and guides to help you in your iOS development journey.

Think about colleagues and friends who could find this guide useful and send them this link through email or private message, so that they can get the cheat sheet together with all the other material I only share with my email list:

https://matteomanferdini.com/parsing-json-in-swift-cheat-sheet/

You can also share this link on your favorite social network.

Thank you.

Matteo