25 NOVEMBER 2019 / COMBINE,SWIFT

Modern Networking in Swift 5 with URLSession, Combine and Codable

Making HTTP requests is one of first things to learn when starting iOS development. Whether you implement networking from scratch, or use Alamofire and Moya, you often end up with a complex and tangled code. Especially, when it comes to requests chaining, running in parallel or cancelling.

Swift 5 system frameworks already provide us with all the tools that we need to write concise networking layer. In this article we'll implement a promise-based networking agent by using vanilla Swift 5 APIs: `Codable`, `URLSession` and the Combine framework. To battle-test our networking layer, we'll practice with several real-world examples that query Github REST API and synchronize the HTTP requests in chain and in parallel.

Implementing Networking Agent

`Agent` is a promise-based HTTP client. It fulfills and configures requests by passing a single `URLRequest` object to it. The agent automatically transforms JSON data into a `Codable` value and returns an `AnyPublisher` instance:

```
import Combine
```

```swift
struct Agent {

    // 1
    struct Response<T> {
        let value: T
        let response: URLResponse
    }

    // 2
    func run<T: Decodable>(_ request: URLRequest, _ decoder: JSONDecoder = JSONDecoder()) -> AnyPub
        return URLSession.shared
            .dataTaskPublisher(for: request) // 3
            .tryMap { result -> Response<T> in
                let value = try decoder.decode(T.self, from: result.data) // 4
                return Response(value: value, response: result.response) // 5
            }
            .receive(on: DispatchQueue.main) // 6
            .eraseToAnyPublisher() // 7
    }
}
```

> The code requires some basic understanding of Combine. Here is the bird's-eye overview of the Swift Combine framework.

1. `Response<T>` carries both parsed value and a `URLResponse` instance. The latter can be used for status code validation and logging.
2. The `run<T>()` method is the single entry point for requests execution. It accepts a `URLRequest` instance that fully describes the request configuration. The decoder is optional in case custom JSON parsing is needed.
3. Create data task as a Combine publisher

3. Create data task as a Combine publisher.

4. Parse JSON data. We have constrained `T` to be `Decodable` in the `run<T>()` method declaration.
5. Create the `Response<T>` object and pass it downstream. It contains the parsed value and the URL response.
6. Deliver values on the main thread.
7. Erase publisher's type and return an instance of `AnyPublisher`.

After implementing the networking core, we are ready to tackle several real-world examples.

Adding HTTP Request

Throughout the article we'll be working with Github REST API. Let's begin by declaring a namespace for it:

```
enum GithubAPI {
    static let agent = Agent()
    static let base = URL(string: "https://api.github.com")!
}
```

> I am touching on the subject in The Power of Namespacing in Swift.

The first endpoint that we implement is list user repositories:

```
extension GithubAPI {


    static func repos(username: String) -> AnyPublisher<[Repository], Error> {
        // 1
        let request = URLRequest(url: base.appendingPathComponent("users/\(username)/repos"))
        // 2
        return agent.run(request)
            .map(\.value)
            .eraseToAnyPublisher()
    }
}

// 3
struct Repository: Codable {
    // Skipping for brevity
}
```

1. Create a `URLRequest` instance, which describes the request. It doesn't need any additional set up, since the HTTP method defaults to GET.
2. Agent executes the request and passes forward the repositories, skipping the response object. We skip response code validation to focus on the happy path.
3. Declare the Github repository model, which conforms to `Codable`, so that we can parse the response JSON.

Making and Cancelling HTTP Request

Let's battle-test our networking agent and fetch the list of Github repositories. To better understand the steps of execution, we print logs to the console:

```
let token = GithubAPI.repos(username: "V8tr")
    .print()
    .sink(receiveCompletion: { _ in },
          receiveValue: { print($0) })
```

The request completes successfully and prints the list of repositories:

```
receive subscription: (TryMap)
request unlimited
receive value: [... the list of repositories]
receive finished
```

We can cancel the request by using `token`:

```
token.cancel()
```

If the request is cancelled, neither value nor error is received. Our code prints cancellation error *after* the stream is terminated:

```
receive subscription: (TryMap)
request unlimited
receive cancel
... Error Domain=NSURLErrorDomain Code=-999 "cancelled" ...
```

Chaining Requests

Another common task is to execute requests one by one. In this example let's fetch user repositories and then the issues for the first repository. We begin by implementing list issue for a repository API:

```
extension GithubAPI {

    static func issues(repo: String, owner: String) -> AnyPublisher<[Issue], Error> {
        let request = URLRequest(url: base.appendingPathComponent("repos/\(owner)/\(repo)/issues"))
        return agent.run(request)
            .map(\.value)
            .eraseToAnyPublisher()
    }
}

struct Issue: Codable {
    // Skipping for brevity
}
```

The below code executes the requests one by one:

```
let me = "V8tr"
let repos = GithubAPI.repos(username: me) // 1
let firstRepo = repos.compactMap { $0.first } // 2
// 3
let issues = firstRepo.flatMap { repo in
    GithubAPI.issues(repo: repo.name, owner: me)
}
// 4
let token = issues.sink(receiveCompletion: { _ in },
                        receiveValue: { print($0) })
```

1. Create a request that fetches user repositories. Note that the request does not fire until we subscribe to it with `sink()`.
2. Transform the request to return first repository only. The use of `compactMap`

   filters out `nil` values.

3. Chain two requests with the help of Combine. The `flatMap` operator transforms a publisher, which returns first repo, into a new one, which returns issues for that repo.
4. Subscribe to the resulting requests chain. This is where the requests are actually fired.

If you are to run this code, you’ll see the issues list printed to debug console.

Let’s take a moment to appreciate how easy it was. The code is a breeze to read. Furthermore, it scales well if we are to add more requests to the chain.

Running Requests in Parallel

When HTTP requests are independent from each other, we can execute them in parallel and combine their results. This speeds up the process, compared to chaining, since the overall loading time equals to the one of the slowest request.

In this section we’ll list repositories and members of an organization in parallel. But before we do that, let’s make a small refactor.

Our `GithubAPI` shares lots of code in common, that can be extracted into a new method:

```
extension GithubAPI {

    static func run<T: Decodable>(_ request: URLRequest) -> AnyPublisher<T, Error> {
        return agent.run(request)
            .map(\.value)
            .eraseToAnyPublisher()
    }

    static func repos(username: String) -> AnyPublisher<[Repository], Error> {
        return run(URLRequest(url: base.appendingPathComponent("users/\(username)/repos"))
    }

    // Skipping `issues(repo:owner:)`
}
```

Now we can add the *list-organization-repositories* and *org-members-list* APIs:

```
extension GithubAPI {

    static func repos(org: String) -> AnyPublisher<[Repository], Error> {
        return run(URLRequest(url: base.appendingPathComponent("orgs/\(org)/repos")))
    }

    static func members(org: String) -> AnyPublisher<[User], Error> {
        return run(URLRequest(url: base.appendingPathComponent("orgs/\(org)/members")))
    }
}
```

Let’s call both requests in parallel and combine their results:

```
let members = GithubAPI.members(org: "apple") // 1
let repos = GithubAPI.repos(org: "apple") // 2
let token = Publishers.Zip(members, repos) // 3
    .sink(receiveCompletion: { _ in },
          receiveValue: { (members, repos) in print(members, repos) }) // 4
```

1. Create members request

1. Create members request.
2. Create repositories request.
3. Create and fire the combined request. We are using Combine’s `Zip` publisher, which waits until both requests have completed and then delivers their results as a tuple.

If you run the code, it will print Apple’s Github members and repositories.

Source Code

Here you can find the final project, which complements this article.

What’s Next

In real world there is more involved into networking. It’s nice to be able to retry requests, persist and automatically renew authorization token, unit test the networking layer, cache responses and much more. These topics are to be covered in individual articles on the subject.

Wrapping Up

Swift 5 is a game changer for networking. The promise-based HTTP agent that we’ve built is just 15 lines of code. It scales well and makes HTTP requests synchronization a breeze.

[illegible]

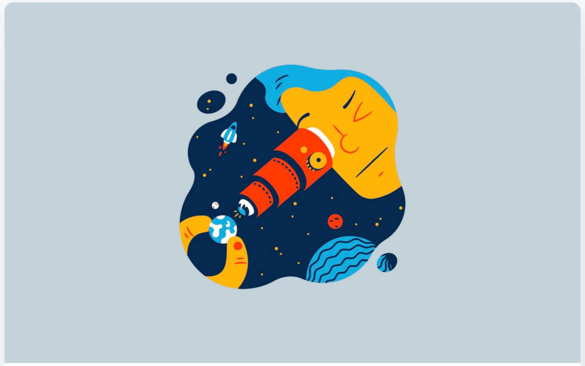
Note that since we are using the Combine framework, the minimal requirements are Swift 5.1, Xcode 11 and iOS 13 (iPadOS).

Thanks for reading!

If you enjoyed this post, be sure to follow me on Twitter to not miss any new content.

Vadim Bulavin
Creator of Yet Another Swift Blog. Lead iOS Engineer at EPAM. Coding for fun since 2008, for food since 2012.

Follow



Error Handling in Swift Combine Framework

See all 6 posts →

property wrappers.

VADIM BULAVIN 6 MIN READ

with the Swift Combine framework.

VADIM BULAVIN 4 MIN READ

Latest publications · Twitter · vadymbulavin@gmail.com