Network Requests and REST APIs in iOS with Swift (Protocol-Oriented Approach)

Networking is a requirement for most modern iOS apps.

Networked apps usually interface with a remote web service that provides the data. And often, this web service is a REST API that returns data in JSON format.

Writing the networking layer of an iOS app, though, is not a simple task. To make asynchronous network calls, you need to use many features of Swift and UIKit, like the URLSession class and the Codable. Moreover, many parts of the app's architecture need to interact, making the task more complicated than it seems.

It's easy to say: *"I need to get some data from a REST API*." But such sentence hides a ton of complexity. Many developers

simply put together pieces of networking code they find on Stack Overflow, o use a networking library.

But networking has a lot of hidden pitfalls.

I once worked on a project where strange bugs happened randomly. The app displayed a list of items, and the user could add more. But sometimes, when adding a new item, the app would reply with an alert saying that the object already existed on the server. Since that was a new item, it clearly was not possible. And the problem was even weirder. The alert would not only show once but multiple times.

After a more in-depth investigation, I discovered that the problem was caused by the networking stack of the app, which had the wrong architecture. Network calls and callbacks were handled through notifications, which I usually recommend to avoid. Since there were many listeners for the same notification, network calls for the same item were duplicated. The server then rejected the extra network calls, causing the multiple alerts to appear in the app.

Architecture is a topic I often cover in my articles because this is the vital foundation of every iOS app. Even if you use the iOS SDK correctly **if you structure your code in the wrong way you end with all sorts of problems in your app**.

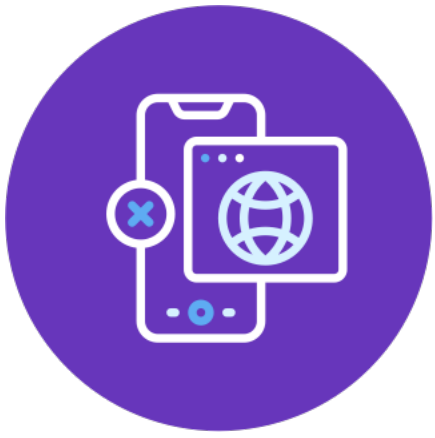
Contents

SECTION 1

The Internet Technologies Behind Remote API Calls

SECTION 2

Making Network Requests in iOS Apps



SECTION 3

Fetching and decoding data

SECTION 4

Protocol-Oriented Network Layer Architecture

SECTION 1:

The Internet Technologies Behind Remote API Calls

Network requests in iOS apps don't happen in a vacuum. Since network requests to REST APIs go through the internet, they rely on protocols and standards you need to understand if your app relies on the network to retrieve its data.

The required steps to perform network requests to a REST API from an iOS app

Performing network requests in an iOS app does not merely amount to adding some extra code. There are many moving parts you need to understand when connecting to a remote web service in iOS. In this article, we will look at each aspect, one by one.

In summary, these are the steps you need to go through to perform a network request in an iOS app:

- **Understand how the remote web service works**.

Nowadays, there are many public APIs on the internet. Each one comes with its implementation and documentation. And, if you work for a company or a client, you might have to interface with a private one. Modern web services often, but not always, follow based on the *REST architecture*.

- **Understand how the HTTP protocol works.**

REST APIs are based on the *HTTP protocol*, which is the communication protocol used by the world wide web. Understanding HTTP means knowing how *URLs* are structured, what actions you can express using *HTTP methods*, how you can express *parameters* in a request, and how to send or receive data.

- **Get a list of all the URLs to make the requests you need**.

Every REST API offers a series of URLs to fetch, create, update, and delete data on a server. These are unique to each API. Their structure depends on the choices made by the developers that created the API. If you are lucky, an API comes with proper documentation. Often though, especially when you interface with a private API, you have to talk to the server-side developers.

- **Learn how to use the URL loading system in the iOS SDK**.

iOS has a robust networking API that addresses all your networking needs, especially for the HTTP protocol. The workhorse of this API is the *URLSession* class. On the internet, you can also find alternative networking libraries, which I don't recommend using.

- **Perform a network request to get the data you need in your app.**

By putting together the elements I listed above, you can finally write the code to perform a remote API call and get back some data. Data comes in different formats, like binary data (for media files), JSON, XML, Markdown, HTML, or others.

Before you can use such data in your app, you have to parse it and convert to your model types. Binary data is usually directly convertible into the appropriate types you find in the iOS SDK. In the case of structured data like JSON, parsing is also quite straightforward. For more complex formats like XML, Markdown, and HTML you could have to write a custom parser.

- **Handle the asynchronous nature of network calls in Swift.**

If making network requests alone was not already hard enough, you have to add to it the fact that you need to run network requests *asynchronously*.

Network requests are inherently slow since it takes time for a server to respond and to transfer data over the network.

That means that networking code needs run in the background, or your app will become unresponsive for long periods. This has different implications in how you write your Swift code, how you handle callbacks, and how you manage memory.

- **Finally, use the retrieved data in your app**.

This is not as straightforward as it sounds. Network requests can fail for many reasons, so you need to handle errors and missing data. You also need to update the UI of your app and show to the user that data is being fetched over the network.

REST APIs use URLs and the HTTP protocol to identify resources and transfer data

You access any REST API through the internet. That means that the various resources offered by the API are identified by a set of uniform resource locators or URLs.

A URL has different components, but in the context of REST APIs, we are usually interested in just three:

- **The host**, which is typically a name (or sometimes an IP address) that identifies the other endpoint (the *server*) we are going to connect to.

- **A path**, which identifies the resource we are looking for.
- **An optional query**, where we can add extra parameters to affect the data we get back (filtering, sorting, paging, etc.).

URLs though, are just a part of what you need to understand to communicate with a REST API. The other part is the Representational State Transfer architecture or REST.

REST is a type of architecture for web services. But as iOS developers, **we don't care how the entire REST architecture works** on the side of the server. All we care about is what we see from an iOS app.

Making REST API calls using HTTP requests

REST works over the Hypertext Transfer Protocol (HTTP), which was created to transmit web pages over the internet. Simply put, in HTTP, we send requests to a server, which sends back responses.

An *HTTP request* usually contains:

- **a URL** identifying the resource we want;
- **an HTTP method** that states the action we want to perform;
- **optional parameters** for the server in the form of HTTP

headers;

- **some optional data** we might want to send to the server.

```
GET /index.html HTTP/1.1
Host: www.example.com
Authorization: Basic YWxhZGRpbjpvcGVuc2VzYW1l
```

Most REST APIs use only a subset of HTTP methods to express which actions you can perform:

- `GET`, to fetch a resource
- `POST`, to create or update a resource
- `DELETE`, to delete a resource

Some APIs can also use the `HEAD`, `PUT` or `PATCH` methods, although it depends on the skills of the API developer. How these work depends on the specific API you are using, so you always have to check the documentation to see if they are available and what they do.

When it comes to parameters, you might have noticed we have two options: either the query string in the URL or the HTTP headers.

So which one should you use?

Details usually depend on the API, but, in general:

- The query string is for parameters related to the resource you are accessing.

- The HTTP headers are for parameters related to the request itself, for example, authentication headers.

Finally, in the optional data section of a request, we put the data we want to send to the API when we are creating or modifying a resource. If you are simply fetching a resource, you don't need to add any data to your request. In fact, the HTTP specification states that a server can reject a `GET` request that contains data.

Most REST APIs return structured JSON and binary data

As I mentioned above, in HTTP, you make requests, and the server replies with responses. An HTTP response usually carries:

- **a status code**, which is a number that tells you if our call was ok or if there was some error;
- **some HTTP headers** specifying extra information about the response;
- **data**, if you requested some.

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Content-Type: text/html; charset=UTF-8
Content-Encoding: UTF-8
Content-Length: 138
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
```

```
Connection: close
```

While there are many formats, a REST API can use, most APIs return data in the Javascript Object Notation (JSON) format. JSON is a data format made to be lightweight, easy for humans to read, and easy for machines to generate and parse.

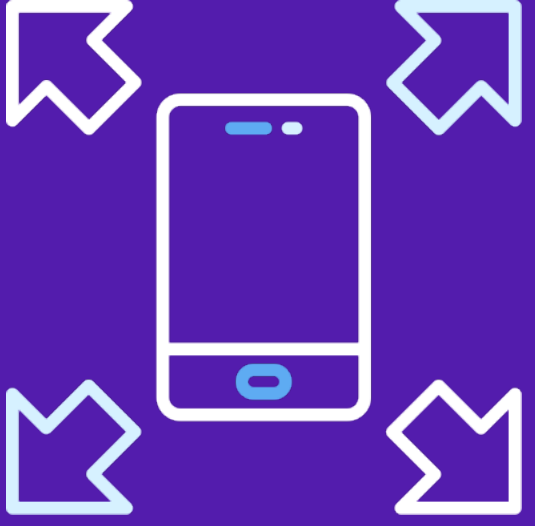
Some web services, though, might use other formats. Common formats are XML, Markdown, or HTML. If you interact with a Windows server, you might receive data in the SOAP format, which requires you to write a custom parser, since it based on XML.

When communicating with remote APIs, we don't only receive structured data. It is common to receive media files like images or videos, which are transmitted as pure binary data.

Beware though that binary data does not come along with the initial response, to keep the latter lightweight. What you get are URLs in the JSON data usually pointing to a content delivery network. That means you will be contacting a different server, with its own rules, so keep it in mind.

SECTION 2:

Making Network Requests in iOS Apps



Like on other platforms, in iOS you don't interact directly with the networking hardware, but rely on a more abstract layer, called *URL Loading System*. Some developers rely on third-party networking libraries, but that's not necessary and comes with several drawbacks.

Should you use a third-party library for iOS networking like AlamoFire or AFNetworking?

When it comes to making network requests in iOS, a high number of developers rely on a networking library like AlamoFire or AFNetworking. Since so many people do it, should you use a library instead of iOS networking API?

My short answer: no.

I'll explain why in the next section. But first, I want to refute some of the reasons why people chose to use such libraries

in the first place (at least, the ones I could find):

- **They are easier to use**.

But are they, really? As I will show you in a moment, you can make network calls in iOS with very little code and using only a couple of classes from the Foundation framework. Yes, the Apple docs for the iOS SDK are a bit terse, but that's a problem of the documentation, not of the API. Third-party libraries also have large documentation, FAQs, migration guides, and many questions on Stack Overflow. They don't look that easier to use.

- **They are asynchronous**.

This is something I don't get. It implies that using the URL loading system in iOS is only synchronous, which is not true. That was not true even with the old NSURLConnection class, which is now deprecated. So I don't understand why people offer this as a benefit.

- **You write less code**.

It depends. It might be true for straightforward network requests, but I would also dispute that. Also, less code does not necessarily mean less complexity, and it also does not necessarily imply time saved. More below.

- **AlamoFire's API uses method chaining**.

This is a nice feature made possible by Swift. The problem

here though is that this coding style, typical of functional reactive programming, forces you into a specific architecture, which is too complex to discuss here.

You also have to decide if this feature alone justifies using a big, third party library. If all you want is method chaining, you can add your implementation on top of the iOS networking API.

- **They reduce boilerplate code in your project**.

No, they don't. The boilerplate just ends somewhere else. The reason is that adopting the approach of a library takes away a lot of the flexibility you have when you can choose your abstractions freely. This will be clearer by the end of the article.

- **You can study them and become a better programmer**.

You can also review them without putting them in your projects. And by the way, I don't know you, but I prefer to spend my learning time on well-written material or conference talks instead of sifting through thousands of lines of undocumented code, trying to understand how it works. For example, try to find your way around this request file in the AlamoFire library.

Why you should not use a third-party library and stick to the

iOS SDK instead

Now, let's see why I recommend not to use networking libraries in your iOS apps.

First of all, this is an opinionated subject, and you will find many opinions on this topic. As they say, opinions are like... well, let's not go there.

In the end, though, it boils down to one of the skills you have to develop as a developer. You need to carefully consider all the pros and cons when deciding whether you should use any library in a project. **Do not just do what someone tells you to do**. And this rule, of course, includes me too.

My biggest concern about using a networking or any third-party library can be summarized in one sentence: **you add a substantial external dependency to your project**. And dependencies always come with costs:

- **You don't own the code in the library.**

If something does not work, you now have a massive chunk of code you need to understand and debug. All that code you didn't write is now, all of a sudden, there for you to sift through. You now need to read a ton of code you didn't write, and you don't know how it works. Code that might also use advanced techniques you don't fully understand.

- **Swift and iOS updates can break the library.**

What happens when the next versions of iOS and Swift come out, and the library stops working? You now depend on an external party to fix it. And that is assuming that the library is well maintained by its developers. Otherwise, you are left on your own.

I worked on projects in which releases had to be delayed because of some libraries that were not going to be updated for new versions of iOS. The team had to spend a lot of time removing the libraries and rewriting all the code that used them.

- **The developers can change how the library works at any time.**

Yes, Apple changes its APIs too. But do you want to depend on more than one third-party besides Apple? At least, Apple gives you time, deprecating APIs with warnings in Xcode, removing them only in future iOS releases. With free, open-source libraries, you have no guarantee. And when a library does evolve, you have to go through migrations you didn't plan for.

- **Libraries force architectural decisions in your project.**

This is something I rarely see mentioned, but to me, it's a big one. I can tell you from direct experience that adding a library to your project often means that you have to work your way around its quirks. You can always refactor your code when its structure does not fit your needs anymore. With a

library, someone else made that decision for you, and you have to live with it.

- **Shortcuts are great until they are not.**

Libraries bring along many pitfalls because let's be honest, some solutions are not thought that well.

AlamoFire has an extension to request images asynchronously through the `UIImageView` class. Everyone loves it because it's so simple to use. Except that you should never make network requests from UI elements. That couples your code to the model of your app and to the networking SDK, which you should avoid.

Have you tried to do that inside table view cells? If you haven't, I'll spare you the time waste and tell you what happens. As you scroll through a table view and cells get reused, asynchronous network callbacks go to the wrong cell objects. You now have to write weird code in your cells to guard against this problem.

- **Libraries make your code harder to test.**

Since a library decides the architecture for you, you often cannot properly structure your code for testing. Granted, you can often refactor you code anyway to be able to write unit tests, but that usually requires more advanced testing techniques and the extensive use of test doubles.

Handling HTTP sessions though the URLSession class

After you understand how the HTTP protocol and REST APIs work, it's time to make network requests from our app.

Many developers find the networking API in iOS hard to use because they don't understand how it works. This is why they often rely on external libraries that "do the work" for them. But as I said, this is a documentation problem, not an SDK problem.

It must be said that knowing the entire iOS URL loading system works can be daunting. Its complexity is justified since the SDK needs to handle many scenarios and protocols. But **the part you need to know to make a network request is quite straightforward**. Once you get it, you can expand your knowledge to include the parts you need.

There are three fundamental types you need to use to make an HTTP request.

The first of the three is URLSession class. A core concept of HTTP is the *session,* which is a sequence of requests and responses to retrieve related data. An easy way to understand the idea is thinking about how your browser loads a web page.

Nowadays, web pages are composed of many parts. The first thing your browser requests is the HTML source code of a page. This contains links to many other resources, like images, videos, CSS style sheets, javascript files, and so on. For the whole page to render, the browser needs to retrieve each resource separately and does so in a single session.

As its name implies, you use the `URLSession` class to manage HTTP sessions. Using the same `URLSession` instance to make multiple requests allows you to share configurations or take advantage of technologies like HTTP/2 Server Push when available.

In practice though using a shared `URLSession` instance across multiple requests requires more advanced architectural concepts that are beside the scope of this article. In most apps, you can get away with using separate instances.

Making a network request using the URLRequest and URLSessionTask classes

The two other necessary classes to perform network requests are URLRequest structure and the URLSessionTask class.

The former encapsulates the metadata of a single request, including the URL, the HTTP method (GET, POST, etc.), the eventual HTTP headers, and so on. For simple GET requests

though you don't need to use this type at all.

The latter of the two performs the actual transfer of data. You usually don't use that class directly, but one of its sub-classes depending on the task you need. The most common one is URLSessionDataTask, which fetches the content of a URL and returns it as a `Data` value.

You usually don't instantiate a data task yourself. The `URLSession` class does it for you when you call one of its `dataTask` methods. But you have to remember to call the `resume()` method on a data task instance, or it won't start.

So, in short, making an HTTP request in iOS boils down to:

1. instantiating and configuring an instance of `URLSession`;
2. creating and setting a `URLRequest` value for your requests, but only when you need some specific parameters. Otherwise, you can use a simple `URL` value.
3. creating and starting a `URLSessionDataTask`, using the `URLSession` instance you created in step 1.

After all the explanation in this article, these three steps require a surprisingly short amount of code:

```
import Foundation

let session = URLSession(configuration: .default, delegate: nil, delega
let url = URL(string: "example.com")!
let task = session.dataTask(with: url, completionHandler: { (data: Data
    // Parse the data in the response and use it
})
task.resume()
```

Do you really need to use a networking library?

SECTION 3:

Fetching and Decoding Data

Once you understand which part of the URL Loading System allow you perform network requests directed at a REST API, it's time to use them in your code effectively. While their usage is simple and spans only a few lines of code, extending such code is ripe with pitfalls.

Creating model types that match the entities of a REST API

In the rest of this article, we will create a simple app to fetch the top question about iOS development on Stack Overflow. You can find the complete Xcode project on GitHub.

No matter what architectural design pattern you use in your app, you always need model types to represent the data and the business logic of an app. That is true whether you use the vanilla version of the MVC pattern, the four-layered version described in Apple's documentation, my Lotus MVC pattern, or any other MVC derivative like MVVM or VIPER (yes, these are all similar to MVC, see my Lotus MVC article for details).

In apps that are connected to the network, you can structure the model layer however you want. But when your data comes from a remote API, this imposes some constraints on your types. **A remote API does not care about how you design your app**. You are the one that has to adapt, so it's better to look at the data the API returns before defining your types.

This is the JSON data of a question coming from the Stack Exchange API:

```
{
  "items": [
    {
      "tags": [
        "ios",
        "cocoa-touch",
        "uikit",
        "uilabel"
      ],
      "owner": {
```

```
        "reputation": 11418,
        "profile_image": "https://www.gravatar.com/avatar/f847f4085ff41
        "display_name": "Stefan",
      },
      "score": 1762,
      "creation_date": 1246179243,
      "title": "Vertically align text to top within a UILabel"
    }
  ]
}
```

I simplified the above JSON code to include only the fields we are interested in. You can see the full response in the documentation.

You can see from the data returned by the API that the owner of a question is returned as a separate object. It makes sense to have a distinct type in our model as well.

```
struct User {
    let name: String?
    let profileImageURL: URL?
    let reputation: Int?
}

struct Question {
    let score: Int
    let title: String
    let date: Date
    let tags: [String]
    let owner: User?
}

struct Wrapper {
    let items: [Question]
}
```

The optional properties are for fields that, according to the documentation, could be missing. Admittedly, a `User` value with all three properties set to `nil` would not be the best thing in a real app, but that's beside the point of this article. These are implementation details that depend on specific

apps.

Notice that we have an extra `Wrapper` structure because, for consistency reasons, the data of every response is wrapped in another JSON object. Keep in mind that this is just a detail of the Stack Exchange API. But we have to take care of it nonetheless.

Decoding the JSON data returned by a REST API using the Codable protocols

We now have to transform the JSON data we get from the API into our model types. Decoding JSON data in Swift has been an annoying task for a long time, with many different approaches and libraries popping up. Some of these libraries followed the functional programming approach, sometimes adding obscure functional concepts and operators like *functors* and *monads*.

Luckily, in Swift 4, the Codable protocols were introduced, which make parsing JSON straightforward. I have been, for a long time, an advocate of putting code that transforms data into model types, since it's part of an app's business logic. So I was quite pleased to see that the Swift core team followed the same approach with `Codable`.

All we have to do is make our types conform to the `Decodable` protocol. And since I named some properties in the `User` and

`Question` types following Swift's conventions, we also need to provide a mapping for those using `CodingKeys` enumerations.

```
struct User {
    let name: String?
    let profileImageURL: URL?
    let reputation: Int?
}

extension User: Decodable {
    enum CodingKeys: String, CodingKey {
        case reputation
        case name = "display_name"
        case profileImageURL = "profile_image"
    }
}

struct Question {
    let score: Int
    let title: String
    let date: Date
    let tags: [String]
    let owner: User?
}

extension Question: Decodable {
    enum CodingKeys: String, CodingKey {
        case score
        case title
        case tags
        case owner
        case date = "creation_date"
    }
}

struct Wrapper: Decodable {
    let items: [Question]
}
```

I recently read an article that recommends keeping model types decoupled from data decoding. The rationale is that it makes your model types and business logic independent from the underlying data. While it has a point, that approach doubles the types in your project and introduces a lot of boilerplate code.

In my experience, that is rarely necessary. Most of the time, in iOS apps, model types and data coincide. As much as I like to keep responsibilities separate in my code, there is no need to over-engineer it for its own sake. But keep that approach in the back of your mind, since it might be useful someday.

A common but not optimal way of making network requests in iOS apps

Now that we have model types to represent the data we receive, we can finally fetch some data from the Stack Exchange API. I will first show you a common approach, which I see often and that you will probably recognize, but is not optimal. Admittedly, I did this too in the past, but now I knew better.

As we have seen above, making HTTP requests using `URLSession` is straightforward. What most developers do is put that code into a network manager/handler/controller class.

```
class NetworkManager {
    func loadQuestions(withCompletion completion: @escaping ([Question]
        let session = URLSession(configuration: .default, delegate: nil
        let url = URL(string: "https://api.stackexchange.com/2.2/questi
        let task = session.dataTask(with: url, completionHandler: { (da
            guard let data = data else {
                completion(nil)
                return
            }
            let wrapper = try? JSONDecoder().decode(Wrapper.self, from:
            completion(wrapper?.items)
        })
        task.resume()
    }
```

```
}
```

In this article, I am not going to cover error handling, which is a topic by itself. We will just transform any error into a `nil` value, which is enough in many apps you write anyway.

So far, so good. The above code works, and you can already use it to fetch data from the API. But You know already that questions are not the only type of data we need to fetch from the API. Even in our little sample app, we need to make a separate network request to fetch the owner's avatar.

A real app would not stop there, though. It would probably have drill-down navigation, going from a screen with a list of questions to another with the details of a selected item. Even if we are still working with questions, we need to make two different network requests for that, because fetching the data of a single question requires different parameters. And that's to say nothing about fetching data for users or answers.

And that's where problems start.

We need to generalize our `NetworkManager` so that we can use it to make all sorts of requests. And since what changes between requests is the *type* of the data we request, the solution is to use Swift generics.

```
struct Wrapper<T: Decodable>: Decodable {
    let items: [T]
}

class NetworkManager {
    func load<T>(url: URL, withCompletion completion: @escaping (T?) ->
```

```
        let session = URLSession(configuration: .ephemeral, delegate: n
        let task = session.dataTask(with: url, completionHandler: { (da
            guard let data = data else {
                completion(nil)
                return
            }
            switch T.self {
            case is UIImage.Type:
                completion(UIImage(data: data) as? T)
            case is Question.Type:
                let wrapper = try? JSONDecoder().decode(Wrapper<Questio
                completion(wrapper?.items[0] as? T)
            case is [Question].Type:
                let wrapper = try? JSONDecoder().decode(Wrapper<Questio
                completion(wrapper?.items as? T)
            default: break
            }
        })
        task.resume()
    }
}
```

Since how we decode data depends on its type, we now need to switch over it to run the appropriate code. Lengthy conditional statements like the `switch` in the code above violate the Open-closed principle of SOLID. You can move into a separate *parser* class, as I often see, but you will just move the problem somewhere else. And the problem gets even worse when you need to configure each network request differently, which adds yet another lengthy conditional statement before the network request code.

Moreover, with such a method, it's not the `NetworkManager` that tells the caller, which is usually a view controller, what type of data the API returns. It's the caller that needs to specify the correct return type, moving the responsibility out of the network layer.

Except that the caller does not get to decide anything. All it

can do is guess the right type, or the network call will fail. This couples the code of view controllers to the internal implementation of the `NetworkManager`.

Some problems can only be addressed by a proper architecture

If you are a developer aware of the Open-closed principle, and many unfortunately are not, you might know some solutions to the above problem. A common one is to put the shared code in a generic method that can be reused and then break a lengthy conditional statement into separate ones.

But that does not work here, either.

```
class NetworkManager {
    static let questionsURL = URL(string: "https://api.stackexchange.co

    func load(url: URL, withCompletion completion: @escaping (Data?) ->
        let session = URLSession(configuration: .ephemeral, delegate: n
        let task = session.dataTask(with: url, completionHandler: { (da
            completion(data)
        })
        task.resume()
    }

    func loadImage(with url: URL, completion: @escaping (UIImage?) -> V
        load(url: url) { data in
            if let data = data {
                completion(UIImage(data: data))
            } else {
                completion(nil)
            }
        }
    }

    func loadTopQuestion(completion: @escaping (Question?) -> Void) {
```

```
        load(url: NetworkManager.questionsURL) { data in
            guard let data = data else {
                completion(nil)
                return
            }
            let wrapper = try? JSONDecoder().decode(Wrapper<Question>.s
            completion(wrapper?.items.first)
        }
    }

    func loadTopQuestions(completion: @escaping ([Question]?) -> Void)
        load(url: NetworkManager.questionsURL) { data in
            guard let data = data else {
                completion(nil)
                return
            }
            let wrapper = try? JSONDecoder().decode(Wrapper<Question>.s
            completion(wrapper?.items)
        }
    }
}
```

Even if we have a generic `load(url:withCompletion:)` method, we still have a lot of code repetition in the other ones.

The problem here is not in the code but the approach. We keep running into different problems because we try to cram code into the `NetworkManager` class, because someone, somewhere, told us that's the way to do it. The solution, instead, is to choose a different architecture for our networking layer.

SECTION 4:

Protocol-Oriented Network Layer Architecture

The standard architectural approaches to the network layer of an iOS app violate common design principles and create code full of repetition, that needs to be constantly changed to make room for new network requests and data types. Following a protocol-oriented approach, we can avoid all these problems.

API resources should be model types

The network manager approach I showed above is not the only one you find online, although it's pretty standard. To solve the problems I have shown you, a common approach is to take out into a separate *resource* structure all the parameters that cause the lengthy conditionals or the code repetition. This resource structure can be then fed to a generic method that makes the network request.

The code is usually along these lines:

```
struct Resource<T> {
    let url: URL
    // Other properties and methods
}

class NetworkManager {
    func load<T>(resource: Resource<T>, withCompletion completion: @esc
        let session = URLSession(configuration: .ephemeral, delegate: n
        let task = session.dataTask(with: resource.url, completionHand
            guard let data = data else {
                completion(nil)
                return
            }
            // Use the Resource struct to parse data
        })
        task.resume()
    }
}
```

That is definitely a step in the right direction, but it's not quite there yet. While better, it still suffers from one problem: the resource structure gets overloaded with a ton of properties and methods to represent all the possible parameters and decode data in different ways.

Mind you; **the problem is not in the number of properties or methods.** The problem is that they are mutually exclusive. For example, a method that decodes binary data into images cannot be used if the resource represented returns JSON data, and vice-versa. This is again some information that the caller needs to understand because the type exposes an interface that must be used in specific-yet-unspecified ways.

This problem is called *interface pollution*, which happens when a type sports methods it does not need. Interface pollution is a symptom of the violation of another SOLID principle, the Interface segregation principle.

The solution here is to split resources into multiple types which can then share a standard interface and functionality through protocol-oriented programming.

Abstracting API resources with protocols, generics, and extensions

Let's start with the resources provided by the REST API. All remote resources, regardless of their type, share a standard interface. A resource has:

- a URL, ending with a path specifying the data we are fetching (for example, a question)
- optional parameters to filter or sort the data in the response;
- an associated model type into which data needs to be converted.

We can specify all these requirements using a protocol. Then, with a protocol extension, we can provide a shared implementation.

```
protocol ApiResource {
    associatedtype ModelType: Decodable
    var methodPath: String { get }
}

extension ApiResource {
    var url: URL {
        var components = URLComponents(string: "https://api.stackexchar
        components.path = methodPath
```

```
        components.queryItems = [
            URLQueryItem(name: "site", value: "stackoverflow"),
            URLQueryItem(name: "order", value: "desc"),
            URLQueryItem(name: "sort", value: "votes"),
            URLQueryItem(name: "tagged", value: "ios")
        ]
        return components.url!
    }
}
```

The `url` computed property assembles the full URL for the resource using the base URL for the API, the `methodPath` parameter, and the various parameters for the query. Notice that the latter are hardcoded for our example, but you can easily transform them into requirements for the `ApiResource` protocol if you want more fine-grained control over them.

Thanks to this protocol, it's now straightforward to create concrete structures for questions, answers, users, or any other type offered by the Stack Overflow API.

In our little sample app, we only need a resource for questions.

```
struct QuestionsResource: ApiResource {
    typealias ModelType = Question
    let methodPath = "/questions"
}
```

Creating generic classes to perform API calls and other network requests

Now that we have a representation of the resources offered by the API, we need actually to make some network requests.

As we have seen, not all our network request are sent to a REST API. Media files are usually kept on a CDN. That means we need to keep our networking code generic and not tied to the `APIResource` protocol we created above.

So, again, we start by analyzing the requirements from the point of view of the caller. A generic network request needs:

- a method to transform the data it receives into a model type;
- a method to start the asynchronous data transfer;
- a callback to pass the processed data back to the caller.

We again express these requirements using a protocol:

```
protocol NetworkRequest: AnyObject {
    associatedtype ModelType
    func decode(_ data: Data) -> ModelType?
    func load(withCompletion completion: @escaping (ModelType?) -> Voic
}
```

Thanks to these requirements, we can then abstract the code that uses `URLSession` to perform the network transfer. We place this code again into a protocol extension:

```
extension NetworkRequest {
    fileprivate func load(_ url: URL, withCompletion completion: @escap
        let session = URLSession(configuration: .default, delegate: nil
        let task = session.dataTask(with: url, completionHandler: { [we
            guard let data = data else {
                completion(nil)
                return
            }
            completion(self?.decode(data))
        })
        task.resume()
    }
}
```

Like the API resources, our concrete network request classes will be based on the `NetworkRequest` protocol, providing the missing pieces defined by the protocol requirements.

The simplest type of network request is the one for images, for which we only need a URL:

```
class ImageRequest {
    let url: URL

    init(url: URL) {
        self.url = url
    }
}

extension ImageRequest: NetworkRequest {
    func decode(_ data: Data) -> UIImage? {
        return UIImage(data: data)
    }

    func load(withCompletion completion: @escaping (UIImage?) -> Void)
        load(url, withCompletion: completion)
    }
}
```

Creating a `UIImage` value from the received `Data` is straightforward. And since we don't need any particular configuration, the `load(withCompletion:)` method of `ImageRequest` can simply call the `load(_:withCompletion:)` method of `NetworkRequest`.

You can see again that this approach allows us to create as many types of requests as we need. All we need to do is add new classes. There is no need to change existing code, respecting the Open-closed principle.

We can now follow the same process and create a class for API requests.

```
class ApiRequest<Resource: ApiResource> {
    let resource: Resource

    init(resource: Resource) {
        self.resource = resource
    }
}

extension ApiRequest: NetworkRequest {
    func decode(_ data: Data) -> [Resource.ModelType]? {
        let wrapper = try? JSONDecoder().decode(Wrapper<Resource.Model
        return wrapper?.items
    }

    func load(withCompletion completion: @escaping ([Resource.ModelType
        load(resource.url, withCompletion: completion)
    }
}
```

The `APIRequest` class uses a generic `Resource` type. The only requirement is that resources conform to `APIResource`. Conforming to the `NetworkRequest` protocol was also not that complicated. Since the API returns JSON data, all we need to do is decode the received `Data` using the `JSONDecoder` class.

We now have an extensible protocol-oriented architecture, which we can expand as we please. We can add new API resources as needed, or new types of network request to send data or download other types of media files.

Performing network requests to populate the user interface of view controllers

We finally reached the last step of this long article, where we

fetch data from the Stack Exchange API, and we display it on screen.

First of all, we need to create the UI for our app in an Xcode storyboard.

To create a scrolling interface, I used a static table view with two cells, composed of simple labels arranged with a few Auto Layout constraints. You can see the details in the full Xcode project.

We then need a custom view controller with a few outlets that connect to our UI elements.

```
class ViewController: UITableViewController {
    @IBOutlet weak var scoreLabel: UILabel!
    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet weak var ownerNameLabel: UILabel!
    @IBOutlet weak var ownerAvatarImageView: UIImageView!
    @IBOutlet weak var ownerReputationLabel: UILabel!
    @IBOutlet weak var askedLabel: UILabel!
    @IBOutlet weak var tagsLabel: UILabel!
}
```

We also need to format the data correctly, adding decimal separators to numbers and transforming dates into a fluent sentence like shown in the app's mockup.

```
extension Int {
    var thousandsFormatting: String {
        let formatter = NumberFormatter()
        formatter.numberStyle = .decimal
        return formatter.string(from: NSNumber(value: self))!
    }
}

extension Date {
```

```
    var timeAgo: String {
        let calendar = Calendar.current
        let units: Set<Calendar.Component> = [.month, .year]
        let components = calendar.dateComponents(units, from: self, to:
        let year = components.year!
        let month = components.month!
        return "\(year) "
            + (year > 1 ? "years" : "year")
            + ", \(month) "
            + (month > 1 ? "months" : "month")
            + " ago"
    }
}
```

To improve the user experience, we can hide the UI while we fetch data from the remote API, showing it only after populating the UI. How you do that depends on the specifics of your user interface. Since we are using a static table view here, we can set the height of both cells to zero while loading.

```
class ViewController: UITableViewController {
    @IBOutlet weak var scoreLabel: UILabel!
    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet weak var ownerNameLabel: UILabel!
    @IBOutlet weak var ownerAvatarImageView: UIImageView!
    @IBOutlet weak var ownerReputationLabel: UILabel!
    @IBOutlet weak var askedLabel: UILabel!
    @IBOutlet weak var tagsLabel: UILabel!

    private var loading = true

    override func tableView(_ tableView: UITableView, heightForRowAt in
        return loading ? 0.0 : UITableView.automaticDimension
    }

    override func tableView(_ tableView: UITableView, estimatedHeightFo
        return UITableView.automaticDimension
    }
}

private extension ViewController {
    func configureUI(with question: Question) {
        scoreLabel.text = question.score.thousandsFormatting
        titleLabel.text = question.title
        set(tags: question.tags)
        ownerNameLabel.text = question.owner?.name
```

```
        ownerReputationLabel.text = question.owner?.reputation?.thousar
        loading = false
        tableView.reloadData()
    }

    func set(tags: [String]) {
        guard !tags.isEmpty else {
            tagsLabel.text = nil
            return
        }
        tagsLabel.text = tags[0] + tags.dropFirst().reduce("") { $0 + '
    }
}
```

Everything is set to finally fetch the data when the view controller loads.

```
class ViewController: UITableViewController {
    @IBOutlet weak var scoreLabel: UILabel!
    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet weak var ownerNameLabel: UILabel!
    @IBOutlet weak var ownerAvatarImageView: UIImageView!
    @IBOutlet weak var ownerReputationLabel: UILabel!
    @IBOutlet weak var askedLabel: UILabel!
    @IBOutlet weak var tagsLabel: UILabel!

    private var loading = true
    private var request: AnyObject?

    override func viewDidLoad() {
        super.viewDidLoad()
        fetchQuestion()
    }

    override func tableView(_ tableView: UITableView, heightForRowAt in
        return loading ? 0.0 : UITableView.automaticDimension
    }

    override func tableView(_ tableView: UITableView, estimatedHeightFo
        return UITableView.automaticDimension
    }
}

private extension ViewController {
    func configureUI(with question: Question) {
        scoreLabel.text = question.score.thousandsFormatting
        titleLabel.text = question.title
        set(tags: question.tags)
```

```
        ownerNameLabel.text = question.owner?.name
        ownerReputationLabel.text = question.owner?.reputation?.thousar
        loading = false
        tableView.reloadData()
    }

    func set(tags: [String]) {
        guard !tags.isEmpty else {
            tagsLabel.text = nil
            return
        }
        tagsLabel.text = tags[0] + tags.dropFirst().reduce("") { $0 + '
    }

    func fetchQuestion() {
        let questionRequest = ApiRequest(resource: QuestionsResource())
        request = questionRequest
        questionRequest.load { [weak self] (questions: [Question]?) in
            guard let questions = questions,
                let topQuestion = questions.first else {
                    return
            }
            self?.configureUI(with: topQuestion)
            if let owner = topQuestion.owner {
                self?.fetchAvatar(for: owner)
            }
        }
    }

    func fetchAvatar(for user: User) {
        ownerAvatarImageView.image = nil
        guard let avatarURL = user.profileImageURL else {
            return
        }
        let avatarRequest = ImageRequest(url: avatarURL)
        self.request = avatarRequest
        avatarRequest.load(withCompletion: { [weak self] (avatar: UIIma
            guard let avatar = avatar else {
                return
            }
            self?.ownerAvatarImageView.image = avatar
        })
    }
}
```

The `fetchQuestion()` method fetches the data for the top questions using an instance of `APIRequest` configured with a `QuestionsResource` value. All we need to do then is to call its

`load(withCompletion:)` method and use the returned array of questions to configure our UI.

Fetching the avatar of the user requires a separate `ImageRequest`. The `fetchAvatar(for:)` method of our `ViewController` class follows the same line as the `fetchQuestion()` method.

The image request can happen only after the first API request is completed since it's the `User` structure that carries the URL for the owner of a question. That is why `fetchAvatar(for:)` only gets called in the callback of the `APIRequest`. We could put the code there, but that would lead to callback hell, so it's better to have a separate method.

Notice one final thing: we need to store each API request in the `request` property while it gets executed. If we don't, ARC will deallocate immediately after we create it and we won't get a callback.

Our app is finally complete. You can run it and see the UI appear with data coming from the StackExchange API.

If you have a fast internet connection, the UI might appear immediately. To see how a networked app works with slow connections, you can use the network link conditioner to slow down your network requests.

Summary

In this article, I showed you not only how to send network requests to a remote REST API, but also how to structure the networking layer in your apps. While the example app we built is simple, you can see that it already involves a lot of complexity. Correctly architecting your networking code is an investment that pays many future dividends, since adding new API calls to an app becomes straightforward, with a higher degree of code reuse.

The important concepts to remember are:

- **A REST API relies on URLs and the HTTP protocol**.

The REST architecture for web services uses URLs to specify resources and parameters and HTTP methods to identify actions. Responses use HTTP status codes to express results and the body of a response to return the requested data, often in JSON format.

- **You don't need a networking library like AlamoFire or AFNetworking**.

External libraries add dependencies and restrictions to your app. Third-party libraries can change or break without notice, and you have to adapt your architecture to the choices made by someone else.

- **You perform network requests in iOS using the URL loading system**.

There are three types you need to perform network re-

quests. The `URLSession` class handles HTTP sessions, the `URLRequest` structure represents a single request in a session, and the `URLSessionTask` class is the type that performs asynchronous data transfers.

- **Monolithic network managers create problems in your code**.

Many developers put all the networking code inside a single manager class. This violates sound principles of software development and creates code that is hard to change, easy to break, and hard to test.

- **Protocol-oriented programming is the best tool to architect the networking layer of your apps**.

Using a combination of protocols, extensions, and concrete types, you can create a flexible hierarchy that is easy to extend with new resources and network requests.

SwiftUI is the future of UI development. And yet, many developers share some misconceptions that prevent them from understanding how SwiftUI works and from using it in real apps.

GET THE FREE BOOK NOW