

---

# Cocoapods, HTTP, Networking, and JSON

---

When building out our applications, there will come times when we want to build a very complicated UI or perform some sort of special task. In these cases, we have two options. We can try to build this new feature from scratch all by ourselves **OR** we could use a library (code) which other people have already written and made public because they figured that other people would want to have this feature in their application.

## What is Cocoapods?

From the cocoa pods website: “CocoaPods is a dependency manager for Swift and Objective-C Cocoa projects. It has over 53 thousand libraries and is used in over 3 million apps.” Let us break down what this means. A **dependency** just refers to a library, or code, that someone else has written and made public for others to use. A **dependency manager** is a tool that manages all the libraries that are used in your application. Just to give some examples for Python application’s dependency manager is pip and node.js application’s dependency manager is npm.

For Swift application’s this is Cocoapods. Some extremely popular and useful Cocoapods that we definitely recommend you guys check out is **Alamofire** (networking library) and **SnapKit** (library for easy AutoLayout). Essentially, we will be using Cocoapods to install and be able to use any external libraries that we want to use in our application.

## Setting up Cocoapods

Before we can use Cocoapods, we must install it on our computers. To do so, go to your terminal and run the following command (NOTE: you should have Xcode installed already beforehand):

```
$ sudo gem install cocoapods
```

Once this command finishes running, you should have cocoapods installed and should not have to reinstall it ever again. Now, in order to setup cocoapods in our Xcode project, use your terminal and navigate to your project’s root directory. Once inside run the following command:

```
$ pod init
```

What this command does is create a boilerplate **Podfile** inside the current directory. Upon opening the Podfile, you should see something like this:

```
platform :ios, '9.0'

# ignore all warnings from all pods
inhibit_all_warnings!

target 'SampleProject' do
  use_frameworks!

  # Pods for SampleProject
  pod 'Alamofire'
  pod 'SnapKit'
end
```

In the snippet above I inserted two cocoapods already as an example but yours shouldn't contain any in the beginning. Essentially, anytime you want to add a new cocoapod to your project, all you have to do is come into this Podfile and insert it as **pod '[name]'**. In this snippet, I am declaring that I want to use the **Alamofire** and **SnapKit** cocoapods. Once you have written all the cocoa pods that you need, save the file and return to the terminal. The next command that we want to run is:

```
$ pod install
```

What this command does is it looks at your **Podfile** (needs to exist in the directory or else this command will not work) to see all the cocoa pods that you want to use, installs all of them, and integrates them with your .xcodproj into a **.xcworkspace**. For example, if we our project was called SampleProject and we had a **SampleProject.xcodproj**, after running **pod install**, we would have a **SampleProject.xcworkspace** in the same directory. After running pod install, you should always be developing in **.xcworkspace** instead of **.xcodproj**. This is because **.xcworkspace** is where you will be able to import and use all the cocoa pods that you installed. If at any point later on you want to install more cocoapods, simply go back to the Podfile, add your cocoa pod, and then run pod install again.

## How do you make an app communicate with the internet?

So far in this course, we've only dealt with hard-coded data that we've made ourselves to show on screen, but most apps don't have static information. **HTTP requests** are a method to communicate between a client (your iOS app) and a server (the internet).

HTTP requests are used all over development, and there are many different types. The most common types of requests are **GET** and **POST** requests. GET requests are used to get information from the internet (ex. getting information from google.com), and POST requests are used to post information onto the internet (ex. submitting a form). Other methods include PUT, DELETE, and more.

In this class, we'll be using **Alamofire**, a Cocoapod, to make HTTP requests. Here's an example of making a GET request to fetch recent posts from The Cornell Daily Sun's website. We'll just print the data we get back.

```

let endpoint = "http://cornellsun.com/wp-json/wp/v2/posts"
Alamofire.request(endpoint, method: .get)
    .validate().responseObject { response in
        // Depending on what response JSON we get here,
        // we can appropriately handle it.
        switch response.result {
            // If the response is a success, print the data
            case let .success(data):
                print(data)
            // If the response is a failure, print the error
            case .failure(let error):
                print(error.localizedDescription)
        }
    }
}

```

The printed response will look something like this:

```

[{"id":
3597603,"date":"2018-07-17T01:45:30","date_gmt":"2018-07-17T05:45:30","guid"
:{"rendered":"http://cornellsun.com/?
p=3597603"},"modified":"2018-07-17T01:45:30","modified_gmt":"2018-07-17T05:4
5:30","slug":"jurassic-world-fallen-kingdom-bites-off-more-than-it-can-
chew","type":"post","link":"http://cornellsun.com/2018/07/17/jurassic-
world-fallen-kingdom-bites-off-more-than-it-can-chew/","title":
{"rendered":"Jurassic World: Fallen Kingdom Bites Off More Than It Can
Chew"},"content":{" ... etc

```

## Interpreting responses from the internet

The response above is formatted in **JSON (JavaScript Object Notation)**, which is a key-value coding format commonly used in server responses. Each JSON is wrapped in two curly braces {}, and data is separated by commas. Values in JSON can be strings, numbers, objects (items wrapped in curly braces), null, booleans, or arrays.

Here's an example of a JSON response you might get:

```

{
    "title":      "HTTP, Networking, and JSON",
    "year_created": 2018,
    "success":    true,
    "topics":     [ "HTTP", "Networking", "JSON" ],
}

```

Keys

Values

This entire thing is an object because it's wrapped between {}.

Usually, the responses will be condensed into one blob of text (as seen in the previous example), so you can use a JSON pretty-printer to format it to make it easier to read. Here's an example of a good JSON printer that might be useful: <https://jsonformatter.org/json-pretty-print>

So, how do you decode these responses and put them on screen?

## Decoding responses from the internet

Starting in Swift 4, there's a protocol called **Codable** that objects, structs, and enums can conform to to make them encodable and decodable from JSON. Codable items can contain an enum for the coding keys of type String and CodingKey in order to decode the values from JSON.

For example, if we wanted to decode the example JSON response into a model of type lecture, we could create a Lecture struct that conforms to Codable as follows:

```
struct Lecture: Codable {  
    var title: String  
    var yearCreated: Int  
    var success: Bool  
    var topics: [String]  
}
```

Notice how the names of our variables match the fields in our JSON. This is done on purpose. With Codable, the names of the variables are expected to be the keys in the JSON

## JSON decoding

In order to decode the JSON response into the Lecture class, we could then create a JSONDecoder, and then call the decode method to convert it from a JSON to a Lecture struct. How cool is that?

```
// Assume we already received the data from an Alamofire call  
let jsonDecoder = JSONDecoder()  
  
// Decode the data as a Lecture (optional type)  
let lecture = try? jsonDecoder.decode(Lecture.self, from: data)
```

**Note:** Since the decoding could throw errors, it's preceded by a try?, which will make the lecture constant an optional. This means that lecture will either contain a Lecture item or be nil.

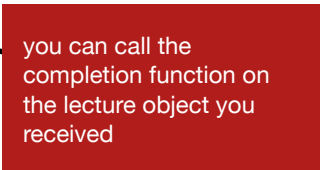
## Using your decoded data: @escaping completions

Once you've decoded the lecture JSON as a Lecture class, how do you use it? Since you've downloaded this JSON from the internet, it might take forever to load, and you only want to display this information on the screen when it's ready (and possibly use a loading indicator in the meantime as the information loads). We also don't want our application to just wait until we get a response to do something else.

This is where a **completion handler** comes in handy. A completion handler is an argument passed into whatever networking function you make that will take in your decoded object(s) and pass them to the ViewController (or wherever you're calling this function) and then run some block of code that you write (i.e. perhaps reload the view with the data on screen).

This completion handler is marked as @escaping because it can escape from the networking function call itself and hand it off to whoever called the function - it doesn't need to go through every line inside the function. Back to the Lecture class example from before: suppose we want to make a function that makes the network call and returns to us the Lecture JSON decoded as a Lecture class. Here's an example of what that would look like:

```
static func getLecture(completion: @escaping (Lecture) -> Void) {  
    Alamofire.request(endpoint, method: .get)  
        .validate().responseJSON { response in  
        switch response.result {  
        case let .success(data):  
            let jsonDecoder = JSONDecoder()  
            if let lecture = try? jsonDecoder.decode(Lecture.self, from:  
data) {  
                completion(lecture)  
            }  
        case .failure(let error):  
            print(error.localizedDescription)  
        }  
    }  
}
```



you can call the completion function on the lecture object you received

Here, you can see that **completion** is a function that takes in one argument which is of type **Lecture** and then has a return type of **Void** (i.e. does not return anything). We only call completion when we know that the response succeeded and we actually have an actual Lecture object.

## Making network calls

It's good practice to create a NetworkManager class with static methods for every network call, with the endpoint urls hidden to the public (in case it's private information).

Then, in your ViewController, you can make a network request and handle it appropriately. For example, if you have a variable for lecture and need to reload the data of a table view after you get your lecture, you would do it like so:

```
func getClasses() {  
    NetworkManager.getLecture { lecture in  
        self.lecture = lecture  
        self.tableView.reloadData()  
    }  
}
```

The { lecture in ... } is the escaping completion we defined in the NetworkManager earlier. In this case, we get the lecture variable only if we made the network call to the endpoint, received a success, received the JSON, and then decoded it. If succeeded, we run this block of code. This ensures that we receive the Lecture object only when it's ready, so the screen isn't frozen while the app is trying to execute all of these network request steps.

## POST requests and query parameters

Sometimes, you want to send information to the network instead of just receiving it. For example, when you search for something (for example, "Taylor Swift") in iTunes, you make a network request by sending the string "Taylor Swift" to the backend in a POST request.

In Alamofire, you can pass in a Parameter object in a network request to the endpoint you choose. The Parameter object is a dictionary mapping strings to strings. For example, in iTunes, the search endpoint requires four parameters: term, country, media, and entity. The search term is the keywords joined by the + sign (so "Taylor Swift" would become "Taylor+Swift").

For example, let's say we're searching songs by Taylor Swift in the US so the media type is music and the entity type is songs. Here's what this would look like as an Alamofire network request (on the next page).

**Note on Alamofire query parameters:** Depending on how your endpoint receives parameters on the backend, you might need to add in another argument to your Alamofire request. For example, if your search endpoint takes in query strings, you'll need to add the extra argument for query string parameter encoding, which is `URLEncoding(destination: .queryString)`. If you're having trouble with your Alamofire POST requests this most likely will fix it! For more information, see this [StackOverflow post here](#).

```
private static let endpoint = "https://itunes.apple.com/search"

static func getTracks(withQuery query: String, completion: @escaping
([Song]) -> Void) {

    let parameters: Parameters = [
        "term" : query.replacingOccurrences(of: " ", with: "+"),
        "country" : "US",
        "media" : "music",
        "entity" : "song"
    ]

    Alamofire.request(endpoint, parameters: parameters)
        .validate().responseJSON { response in

        // handle response here
        switch response.result {
            ...
        }
    }
}
```