iOS Apps with REST APIs

Building Web-Driven Apps in Swift

Christina Moulton

This book is for sale at http://leanpub.com/iosappswithrest

This version was published on 2018-06-26

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

3. Swift JSON Parsing & Networking Calls 101

Nearly every app these days consumes or creates content through a web-based API. In this book we'll mostly use Alamofire[1], a rich networking library, to interact with web services but you can also use iOS's `URLSession` to make REST calls.

3.1 REST API Calls with URLSession

The function to use to make an asynchronous URL request is part of `URLSession`:

```
func dataTask(with request: URLRequest, completionHandler:
  @escaping (Data?, URLResponse?, Error?) -> Void) -> URLSessionDataTask
```

It takes a request which contains the URL. When that request is done or has an error to report it calls the completion handler. The completion handler is where we can work with the results of the call: error checking, saving the data locally, updating the UI, etc.

The simplest case is a GET request. To figure out how to do that we'll need an API to hit. Fortunately there's super handy JSONPlaceholder[2]:

> "JSONPlaceholder is a fake online REST API for testing and prototyping. It's like image placeholders but for web developers."

JSONPlaceholder has a handful of resources similar to what you'll find in a lot of apps: users, posts, photos, albums, ... We'll stick with todos.

> You'll need to create an Xcode project to run the demo code in this chapter. To do that, open Xcode and select File -> New -> Project. Choose a Single View Application. Give it a name (maybe Grok101), make sure it's using Swift (not Objective-C), and choose where you want to save it.

First, let's print out the title of the first todo (assuming that there are todos, which this dummy API already has). To get a single todo, we need to make a GET call to the `todos` endpoint with an ID number. Checking out https://jsonplaceholder.typicode.com/todos/[3] we can see that the id for the first todo is 1. So let's grab it.

[1] https://github.com/Alamofire/Alamofire
[2] https://jsonplaceholder.typicode.com
[3] https://jsonplaceholder.typicode.com/todos/

3.1.1 Creating a URL Request

First, set up the URL request:

```
let todoEndpoint = "https://jsonplaceholder.typicode.com/todos/1"
guard let url = URL(string: todoEndpoint) else {
  print("Error: cannot create URL")
  return
}
let urlRequest = URLRequest(url: url)
```

> Not sure where to put that code? If you created a test project at the start of this chapter, open the `ViewController.swift` file. You can put the test code in `viewDidLoad` like this:
>
> ```
> import UIKit
>
> class ViewController: UIViewController {
>   override func viewDidLoad() {
>     super.viewDidLoad()
>     // test code goes here like this:
>     let todoEndpoint = "https://jsonplaceholder.typicode.com/todos/1"
>     guard let url = URL(string: todoEndpoint) else {
>       print("Error: cannot create URL")
>       return
>     }
>     let urlRequest = URLRequest(url: url)
>     // ... keep adding test code here
>   }
> }
> ```
>
> Now your code will be run when that view controller is shown, which happens right after your app launches.

The `guard` statement lets us check that the URL we've provided is valid.

Then we need a `URLSession` to use to send the request. We can use the default shared session:

```
let session = URLSession.shared
```

Then create the data task:

```
let task = session.dataTask(with: urlRequest, completionHandler: { _, _, _ in })
```

`{ _, _, _ in }` looks funny but it's just an empty completion handler. Since we just want to execute our request and not deal with the results yet, we can specify an empty completion handler here. It needs to have input arguments that match the type of completion handler that's expected, hence the `_, _, _` placeholders for those three arguments.

And finally send it (yes, this is an oddly named function):

```
task.resume()
```

All together it looks like this:

```
// set up url request
let todoEndpoint = "https://jsonplaceholder.typicode.com/todos/1"
guard let url = URL(string: todoEndpoint) else {
  print("Error: cannot create URL")
  return
}
let urlRequest = URLRequest(url: url)

// create and send task
let session = URLSession.shared
let task = session.dataTask(with: urlRequest, completionHandler: { _, _, _ in })
task.resume()
```

Calling this now will hit the URL (from the `urlRequest`) and obtain the results (using a GET request since that's the default). To actually get the results to do anything useful we need to implement the completion handler.

3.1.2 Completion Handlers

Completion handlers can be a bit confusing the first time you run into them. On the one hand, they're a variable or argument but, on the other hand, they're a chunk of code. They can seem odd if you're not used to that kind of thing which is called a closure:

> "Closures are self-contained blocks of functionality that can be passed around and used in your code." - iOS Developer Documentation on Closures[4]

Completion handlers are useful when your app is doing something that might take a little while, like making an API call, and you need to do something when that task is done, like updating the user interface to show the data that you just received. You'll see completion handlers in Apple's APIs like `dataTask(with request: completionHandler:)` and later on we'll add some of our own completion handlers when we're writing functions to make our API calls.

In `dataTask(with request: completionHandler:)` the completion handler argument has a signature like this:

[4] https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Closures.html

```
completionHandler: (Data?, URLResponse?, Error?) -> Void
```

The completion handler takes as input a chunk of code with three arguments: `(Data?, URLResponse?, Error?)` that returns nothing: `Void`.

To specify a completion handler we can write the closure inline like this:

```
let task = session.dataTask(with: urlRequest,
  completionHandler: { data, response, error in
  // this is where the completion handler code goes
})
task.resume()
```

The code for the completion handler is the bit between the curly brackets. Notice that the three arguments in the closure `data, response, error` match the arguments in the completion handler declaration: `Data?, URLResponse?, Error?`. You can specify the types explicitly when you create your closure but it's not necessary because the compiler can figure it out:

```
let task = session.dataTask(with: urlRequest, completionHandler:{
  (data: Data?, response: URLResponse?, error: Error?) in
  // this is where the completion handler code goes
  if let response = response {
    print(response)
  }
  if let error = error {
    print(error)
  }
})
task.resume()
```

Somewhat confusingly, you can actually drop the `completionHandler:` bit and just tack the closure on at the end of the function call. It's called trailing closure syntax[5]. This is totally equivalent to the code above and far more commonly in Swift:

[5] https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/Closures.html#//apple_ref/doc/uid/TP40014097-CH11-ID102

```
let task = session.dataTask(with: urlRequest) { data, response, error in
  // this is where the completion handler code goes
  if let response = response {
    print(response)
  }
  if let error = error {
    print(error)
  }
}
task.resume()
```

You can use a trailing closure whenever the last argument for a function is a closure. We'll be using trailing closure syntax in the rest of our code.

If you want to ignore some arguments you can tell the compiler that you don't want them by replacing them with _ like we did earlier when we weren't ready to implement the completion handler yet. For example, if we only need the `data` and `error` arguments but not the `response` in our completion handler:

```
let task = session.dataTask(with: urlRequest) { data, _, error in
  // can't do print(response) since we don't have response
  if let error = error {
    print(error)
  }
}
task.resume()
```

We can also declare the closure as a variable then pass it in when we call `session.dataTask(with:)`. That's handy if we want to use the same completion handler for multiple tasks. We will use this technique when implementing an OAuth 2.0 login flow since it has lots of steps but we will want to use the same completion handler to handle failure at any step.

Here's how you can use a variable for a completion handler:

```
// create completion handler
let myCompletionHandler: (Data?, URLResponse?, Error?) -> Void = {
  (data, response, error) in
  // this is where the completion handler code goes
  if let response = response {
    print(response)
  }
  if let error = error {
    print(error)
  }
}

let task = session.dataTask(with: urlRequest, completionHandler: myCompletionHandler)
task.resume()
```

So when we run that code what will happen to the closure that we pass in? You might be surprised that it won't get called right away when we call `dataTask(with request: completionHandler:)`. That's a good thing, if it were called immediately then we wouldn't have the results of the web service call yet. Somewhere in Apple's implementation of that function it will get called like this:

```
func dataTask(with request: URLRequest,
  completionHandler: @escaping (Data?, URLResponse?, Error?) -> Void
  ) -> URLSessionDataTask {
  // send the URL request
  // wait for results
  // check for errors and do other processing of the response
  completionHandler(data, response, error)
  // return the data task
}
```

You don't need to write that in your own code, it's already implemented in `dataTask(with: completionHandler:)`. In fact, there are probably a few calls to `completionHandler` for handling success and error cases. The completion handler will just sit around waiting to be called whenever `dataTask(with: completionHandler:)` is done.

So what's the point of completion handlers? Well, we can use them to take action when something is done. For example, here we could set up a completion handler to print out the results and any potential errors so that we can make sure our API call worked. Let's go back to our `dataTask(with request: completionHandler:)` example and implement a useful completion handler.

3.1.3 Handling the Response

Within the completion handler we have access to three arguments: the data returned by the request, the URL response, and an error (if one occurred). We'll check for errors and figure out how to get at the data that we want: the first todo's title. We need to:

1. Check for any errors
2. Make sure we got data
3. Try to transform the data into JSON
4. Access the todo object in the JSON and print out the title

Here's how we'll do all that:

```
let task = session.dataTask(with: urlRequest) {
  (data, response, error) in
  // check for any errors
  guard error == nil else {
    print("error calling GET on /todos/1")
    print(error!)
    return
  }
  // make sure we got data
  guard let responseData = data else {
    print("Error: did not receive data")
    return
  }
  // parse the result as JSON, since that's what the API provides
  do {
    guard let todo = try JSONSerialization.jsonObject(with: responseData, options: [])
      as? [String: Any] else {
      print("error trying to convert data to JSON dictionary")
      return
    }
    // now we have the todo
    // let's just print it to prove we can access it
    print("The todo is: " + todo.description)

    // the todo object is a dictionary
    // so we just access the title using the "title" key
    // so check for a title and print it if we have one
    guard let todoTitle = todo["title"] as? String else {
      print("Could not get todo title from JSON")
      return
    }
    print("The title is: " + todoTitle)
  } catch  {
    print("error trying to convert data to JSON")
    return
  }
}
task.resume()
```

Which prints out:

```
The todo is: {
    completed = 0;
    id = 1;
    title = "delectus aut autem";
    userId = 1;
}
The title is: delectus aut autem
```

In the next section we'll dig into how the JSON parsing works but let's take a quick look at the error checking code first.

First, we check if we received an error using `guard error == nil else { ... }`. For now we're just printing errors and returning, in later chapters we'll build in some error handling.

It's usually best to avoid force unwrapping any optionals using `!` since they will crash if the optional is nil. But we're choosing using one when handling an error:

```
guard error == nil else {
  print("error calling POST on /todos/1")
  print(error!)
  return
}
```

`print(error!)` won't cause a crash because we just checked that `error` isn't nil.

We could do the same thing using `if let` to avoid the force unwrap:

```
if let error = error {
  print("error calling POST on /todos/1")
  print(error)
  return
}
```

If we use `if let` then the compiler won't check that we remembered to include `return` in that block. In other words, this will compile:

```
if let error = error {
  print("error calling POST on /todos/1")
  print(error)
}
```

If we use `guard` then the compiler will make sure that we won't forget that `return` statement. This won't compile:

```
guard error == nil else {
  print("error calling POST on /todos/1")
  print(error!)
}
```

So we have a trade-off to make: we can either avoid force unwrapping or we can have the compiler make sure we include the `return` statement. Either choice is reasonable. I prefer to use `guard` with force unwrapping because I would rather have a crash if I wrote code that isn't right instead of accidentally proceeding to try to parse the JSON when there was an error.

After checking the `error`, we're checking that we did receive data in the response using `guard let responseData = data else { ... }`. If neither of those guard statements find an issue, we proceed to trying to parse the JSON.

3.1.4 Simple JSON Parsing

In this code we're using `JSONSerialization.jsonObject(with: options:)` to convert the `data` to JSON. That's the older way of converting data to JSON. In some cases, it's quicker and easier than using the new way with `Codable`.

As of Swift 4, we can also use `Codable` to create objects or structs from the JSON in the response. We'll look at handling JSON again when we create a struct to represent the todo items. Then in a later chapter, we'll come back to it again to figure how to handle dates and more complex structures.

`JSONSerialization` is easier to use if you only want to pull out one or two items from the JSON. As well, sometimes it's easier to work with `JSONSerialization` if the structure of the objects in your app isn't very similar to the JSON that you receive, though you can work around that with `Codable`. `Codable` is still relatively new and, while it's pretty powerful, there are situations where it can be frustrating to get it working with the JSON that you need to parse.

Here's how we're extracting the title of the todo item from the JSON:

```
do {
  guard let todo = try JSONSerialization.jsonObject(with: responseData, options: [])
    as? [String: Any] else {
    print("error trying to convert data to JSON dictionary")
    return
  }
  // now we have the todo
  // let's just print it to prove we can access it
  print("The todo is: " + todo.description)

  // the todo object is a dictionary
  // so we just access the title using the "title" key
  // so check for a title and print it if we have one
  guard let todoTitle = todo["title"] as? String else {
```

```
    print("Could not get todo title from JSON")
    return
  }
  print("The title is: " + todoTitle)
} catch  {
  print("error trying to convert data to JSON")
  return
}
```

`JSONSerialization.data(withJSONObject: options:)` converts the data that we've received into JSON represented as dictionaries and arrays.

At the top level, the JSON parsing is wrapped in a `do-catch` statement:

```
do {
  // ...
} catch  {
  print("error trying to convert data to JSON")
  return
}
```

That `do-catch` statement with `try` inside it is necessary because `JSONSerialization.data(withJSONObject: options:)` can throw an error[6].

The `catch` statement will catch anything that isn't valid JSON. In other words, any time `JSONSerialization.jsonObject(with: , options:)` can't convert the `responseData` into a valid `Any`. It uses `Any` instead of a more specific type because the top-level JSON could be either a dictionary or an array.

Since the JSON parsing gives us an `Any` but we expect it to be a dictionary, we immediately try to cast it using `as? [String: Any]`. If that cast fails then we print an error message and return:

```
guard let todo = try JSONSerialization.jsonObject(with: responseData, options: [])
  as? [String: Any] else {
  print("error trying to convert data to JSON dictionary")
  return
}
```

Then we can extract the title from the `todo` dictionary using the `title` key and cast that element to `String`:

[6] https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/ErrorHandling.html

```
guard let todoTitle = todo["title"] as? String else {
  print("Could not get todo title from JSON")
  return
}
print("The title is: " + todoTitle)
```

If you just need to access a few elements of the JSON then using `JSONSerialization` in this way can be at least as simple as `Codable`. That code is a little verbose but if you just need a quick GET call to an API without authentication, that'll do it.

3.1.5 Other HTTP Methods

If you need to use a HTTP method type other than GET then you can set the HTTP method in the `URLRequest`. You'll have to do that after the `URLRequest` is created since it's not part of the initializer, so declare the `URLRequest` with `var`, not `let`. Set the code you've been working on aside (you can just comment it out) so we can create a POST request instead:

```
let todosEndpoint = "https://jsonplaceholder.typicode.com/todos"
guard let todosURL = URL(string: todosEndpoint) else {
  print("Error: cannot create URL")
  return
}
var todosUrlRequest = URLRequest(url: todosURL)
todosUrlRequest.httpMethod = "POST"
```

Similar to parsing the JSON in the response, we can use `JSONSerialization` to convert the `newTodo` dictionary to JSON represented as `Data`. Then we set that data as the request's `httpBody` to include it in the request. In this case, the function to use is `JSONSerialization.data(withJSONObject:, options:)`. Like parsing, it can throw an error so we call it using `try` and wrap it in a `do-catch` statement:

```
let newTodo: [String: Any] = ["title": "My First todo", "completed": false, "userId": 1]
let jsonTodo: Data
do {
  jsonTodo = try JSONSerialization.data(withJSONObject: newTodo, options: [])
  todosUrlRequest.httpBody = jsonTodo
} catch {
  print("Error: cannot create JSON from todo")
  return
}
```

Later we'll use `urlRequest.httpBody` again to send JSON that we've converted to `Data` using `Codable`.

Now we can execute this POST request:

```
let session = URLSession.shared
let task = session.dataTask(with: todosUrlRequest) { _, _, _ in }
task.resume()
```

If it's working correctly then we should get our todo back as a response along with the id number assigned to it. Since it's just for testing, JSONPlaceholder will let you do all sorts of REST requests (GET, POST, PUT, PATCH, DELETE and OPTIONS) but it won't actually change the data based on your requests. So when we send this POST request, we'll get a response with an ID to confirm that we did it right but it won't actually be kept in the database so we can't access it on subsequent calls. We can use the same error checking and parsing that we used with our GET request to make sure the API call worked:

```
let todosEndpoint = "https://jsonplaceholder.typicode.com/todos"
guard let todosURL = URL(string: todosEndpoint) else {
  print("Error: cannot create URL")
  return
}
var todosUrlRequest = URLRequest(url: todosURL)
todosUrlRequest.httpMethod = "POST"
let newTodo: [String: Any] = ["title": "My First todo", "completed": false, "userId": 1]
let jsonTodo: Data
do {
  jsonTodo = try JSONSerialization.data(withJSONObject: newTodo, options: [])
  todosUrlRequest.httpBody = jsonTodo
} catch {
  print("Error: cannot create JSON from todo")
  return
}

let session = URLSession.shared

let task = session.dataTask(with: todosUrlRequest) {
  (data, response, error) in
  guard error == nil else {
    print("error calling POST on /todos/1")
    print(error!)
    return
  }
  guard let responseData = data else {
    print("Error: did not receive data")
    return
  }

  // parse the result as JSON, since that's what the API provides
  do {
    guard let receivedTodo = try JSONSerialization.jsonObject(with: responseData,
```

```
      options: []) as? [String: Any] else {
        print("Could not get JSON from responseData as dictionary")
        return
    }
    print("The todo is: " + receivedTodo.description)

    guard let todoID = receivedTodo["id"] as? Int else {
      print("Could not get todoID as int from JSON")
      return
    }
    print("The ID is: \(todoID)")
  } catch  {
    print("error parsing response from POST on /todos")
    return
  }
}
task.resume()
```

In this case we're interested in the `id` property which is an integer so we can extract it from the JSON dictionary using `receivedTodo["id"] as? Int`.

Deleting is pretty similar, except we don't need to set the `httpBody` using a new todo item. To see whether the delete call succeeds, we can just check whether we get an error by using `guard let _ = data`. That will check whether `data` has a value. It's equivalent to `guard data != nil`:

```
let firstTodoEndpoint = "https://jsonplaceholder.typicode.com/todos/1"
var firstTodoUrlRequest = URLRequest(url: URL(string: firstTodoEndpoint)!)
firstTodoUrlRequest.httpMethod = "DELETE"

let session = URLSession.shared

let task = session.dataTask(with: firstTodoUrlRequest) {
  (data, response, error) in
  guard let _ = data else {
    print("error calling DELETE on /todos/1")
    return
  }
  print("DELETE ok")
}
task.resume()
```

That's how to call a REST API from Swift using `URLSession` and `URLRequest`. The code to make the calls themselves is fairly verbose and the level of abstraction is low: you're thinking about todos but having to code in terms of HTTP requests and data tasks. Alamofire[7] is a nice way to get rid of some of the verbosity and work at a higher level of abstraction:

[7] https://github.com/Alamofire/Alamofire

```
let todoEndpoint = "https://jsonplaceholder.typicode.com/todos/1"
Alamofire.request(todoEndpoint)
  .responseJSON { response in
    // get errors
    if let error = response.result.error {
      print(error)
    }
    // get serialized data (i.e., JSON)
    if let value = response.result.value {
      print(value)
    }
    // get raw data
    if let data = response.data {
      print(data)
    }
    // get HTTPURLResponse
    if let httpResponse = response.response {
      print(httpResponse)
    }
}
```

In the next section we'll work out how to make the same requests we did using Alamofire so we that can use that more concise syntax.

Grab the code on GitHub: REST gists[8]

3.2 REST API Calls with Alamofire

In the last section we looked at getting access to REST APIs in iOS using `URLSession`. That approach using `dataTask(with request: completionHandler:)` works just fine for simple cases, like a URL shortener. But these days lots of apps have tons of web service calls that are just begging for better handling: a higher level of abstraction, concise syntax, simpler streaming, pause/resume, progress indicators, ... In Swift those features are available in the Alamofire[9] library.

Let's see how the GET call that we set up previously looks with Alamofire.

First add Alamofire v4.7 to your project using CocoaPods. (See A Brief Introduction to CocoaPods if you're not sure how. Don't forget to close the `xcodeproj` and open the `xcworkspace` instead.)

After adding the pod, you'll need to import Alamofire into the file where you're working:

[8] https://gist.github.com/cmoulton/7ddc3cfabda1facb040a533f637e74b8
[9] https://github.com/Alamofire/Alamofire

```
import UIKit
import Alamofire

class ViewController: UIViewController {
  // ...
}
```

Now we can set up the request:

```
let todoEndpoint = "https://jsonplaceholder.typicode.com/todos/1"
Alamofire.request(todoEndpoint)
  .responseJSON { response in
    // ...
  }
```

We're telling Alamofire to set up & send an asynchronous request to todoEndpoint (without the ugly call to `URL` to wrap up the string). We don't have to explicitly say it's a GET request since that's the default HTTP method. If we wanted to specify the HTTP method then we'd use a member of Alamofire's `HTTPMethod` enum, which includes `.get`, `.post`, `.patch`, `.options`, `.delete`, etc. We can add the method when creating the request to make it explicit:

```
let todoEndpoint = "https://jsonplaceholder.typicode.com/todos/1"
Alamofire.request(todoEndpoint, method: .get)
  .responseJSON { response in
    // ...
  }
```

After making the request, we get the data (asynchronously) as JSON in the `.responseJSON`. We could also use `.response` (for the `HTTPURLResponse`), `.responseData`, or `.responseString`. We can even chain multiple `.responseX` methods, which is often handy for debugging:

```
let todoEndpoint = "https://jsonplaceholder.typicode.com/todos/1"
Alamofire.request(todoEndpoint)
  .responseJSON { response in
    // handle JSON
  }
  .responseString { response in
    if let error = response.result.error {
      print(error)
    }
    if let value = response.result.value {
      print(value)
    }
  }
```

That's neat but right now we just want to get the todo's title from the JSON. We'll make the request then handle it with `.responseJSON`. Like last time we need to do some error checking:

1. Check for an error returned by the API call
2. If no API call error, see if we got any JSON results
3. Check for an error in the JSON transformation
4. If no JSON parsing error, access the todo object in the JSON and print out the title

`.responseJSON` takes care of some of the boilerplate we had to write earlier. It makes sure we got response data then calls `JSONSerialization.jsonObject` for us. So we can just check that we got the JSON object that we expected. In this case, that's a dictionary so we 'll use `as? [String: Any]` in our guard statement.

When using the `.responseX` handlers in Alamofire, the value that you want (e.g., a string for `.responseString`) will be in `response.result.value`. If that value can't be parsed or there's a problem with the call then you'll get an error in `response.result.error`.

So to check for errors then get the JSON if there are no errors:

```
let todoEndpoint = "https://jsonplaceholder.typicode.com/todos/1"
Alamofire.request(todoEndpoint)
  .responseJSON { response in
    guard let json = response.result.value as? [String: Any] else {
      print("didn't get todo object as JSON from API")
      if let error = response.result.error  {
        print("Error: \(error)")
      }
      return
    }
    print(json)
  }
```

You can use `response.result.isSuccess` if you just need to know whether the call succeeded or failed:

```
let todoEndpoint = "https://jsonplaceholder.typicode.com/todos/1"
Alamofire.request(todoEndpoint)
  .responseJSON { response in
    guard response.result.isSuccess else {
      // handle failure
      return
    }
    // handle success
}
```

There's another possibility that our current code doesn't consider well: what if we didn't get an error but we also didn't get any JSON or the JSON isn't a dictionary? Some APIs will return error messages as JSON that's a different format than what you requested. In those cases we need to differentiate between not getting anything and not getting the JSON that we expected. Let's split up the code that checks that we got the JSON we expected and the code that checks `response.result.error` into two separate guard statements:

```
let todoEndpoint = "https://jsonplaceholder.typicode.com/todos/1"
Alamofire.request(todoEndpoint)
  .responseJSON { response in
    // check for errors
    guard response.result.error == nil else {
      // got an error in getting the data, need to handle it
      print("error calling GET on /todos/1")
      print(response.result.error!)
      return
    }

    // make sure we got some JSON since that's what we expect
    guard let json = response.result.value as? [String: Any] else {
      print("didn't get todo object as JSON from API")
      if let error = response.result.error {
        print("Error: \(error)")
      }
      return
    }

    print(json)
}
```

Finally, we can extract the title from the JSON dictionary just like we did in the previous section:

```
let todoEndpoint = "https://jsonplaceholder.typicode.com/todos/1"
Alamofire.request(todoEndpoint)
  .responseJSON { response in
    // check for errors
    guard response.result.error == nil else {
      // got an error in getting the data, need to handle it
      print("error calling GET on /todos/1")
      print(response.result.error!)
      return
    }

    // make sure we got some JSON since that's what we expect
    guard let json = response.result.value as? [String: Any] else {
      print("didn't get todo object as JSON from API")
```

```
      if let error = response.result.error {
        print("Error: \(error)")
      }
      return
    }

    // get and print the title
    guard let todoTitle = json["title"] as? String else {
      print("Could not get todo title from JSON")
      return
    }
    print("The title is: " + todoTitle)
}
```

To POST, we just need to change the HTTP method and provide the todo item as JSON in the `parameters` argument of `Alamofire.request()`:

```
let todosEndpoint = "https://jsonplaceholder.typicode.com/todos"
  let newTodo: [String: Any] = ["title": "My First Post", "completed": 0, "userId": 1]
  Alamofire.request(todosEndpoint, method: .post, parameters: newTodo,
    encoding: JSONEncoding.default)
    .responseJSON { response in
      guard response.result.error == nil else {
        // got an error in getting the data, need to handle it
        print("error calling POST on /todos/1")
        print(response.result.error!)
        return
      }
      // make sure we got some JSON since that's what we expect
      guard let json = response.result.value as? [String: Any] else {
        print("didn't get todo object as JSON from API")
        if let error = response.result.error {
          print("Error: \(error)")
        }
        return
      }
      guard let todoID = json["id"] as? Int else {
        print("Could not get todoID as int from JSON")
        return
      }
      print("The ID is: \(todoID)")
  }
```

And DELETE is nice and compact:

```
let firstTodoEndpoint = "https://jsonplaceholder.typicode.com/todos/1"
Alamofire.request(firstTodoEndpoint, method: .delete)
  .responseJSON { response in
    guard response.result.error == nil else {
      // got an error in getting the data, need to handle it
      print("error calling DELETE on /todos/1")
      print(response.result.error!)
      return
    }
    print("DELETE ok")
}
```

Grab the example code on GitHub[10].

So that's one step better on our journey to nice, clean REST API calls. But we're still interacting with untyped JSON which can easily lead to errors. Next, we'll take another step towards cleaner code by using an Alamofire Router to create the URL requests for us. Then we'll look at how `Codable` can make it easier to convert JSON to objects or structs.

[10] https://gist.github.com/cmoulton/9591be2b10043e6811a845f6dcbe821a