SAMPLE CHAPTER

Swift IN DEPTH

Tjeerd in 't Veen

MANNING

Swift in Depth

by Tjeerd in 't Veen

Chapter 11

brief contents

11 Asynchronous error handling with Result

This chapter covers

- Learning about the problems with Cocoa style error handling
- Getting an introduction to Apple's `Result` type
- Seeing how `Result` provides compile-time safety
- Preventing bugs involving forgotten callbacks
- Transforming data robustly with `map`, `mapError`, and `flatMap`
- Focusing on the happy path when handling errors
- Mixing throwing functions with `Result`
- Learning how `AnyError` makes `Result` less restrictive
- How to show intent with the `Never` type

You've covered a lot of Swift's error handling mechanics, and you may have noticed in chapter 6 that you were throwing errors synchronously. This chapter focuses on handling errors from asynchronous processes, which is, unfortunately, an entirely different idiom in Swift.

Asynchronous actions could be some code running in the background while a current method is running. For instance, you could perform an asynchronous API call to fetch JSON data from a server. When the call finishes, it triggers a callback giving you the data or an error.

Swift doesn't yet offer an official solution to asynchronous error handling. According to rumor, Swift won't offer one until the async/await pattern gets introduced somewhere around Swift version 7 or 8. Luckily, the community seems to favor asynchronous error handling with the `Result` type (which is reinforced by Apple's inclusion of an unofficial `Result` type in the Swift Package Manager). You may already have worked with the `Result` type and even implemented it in projects. In this chapter, you'll use one offered by Apple, which may be a bit more advanced than most examples found online. To get the most out of `Result`, you'll go deep into the rabbit hole and look at propagation, so-called monadic error handling, and its related `AnyError` type. The `Result` type is an enum like `Optional`, with some differences, so if you're comfortable with `Optional`, then `Result` should not be too big of a jump.

You'll start off by exploring the `Result` type's benefits and how you can add it to your projects. You'll create a networking API, and then keep improving it in the following sections. Then you'll start rewriting the API, but you'll use the `Result` type to reap its benefits.

Next, you'll see how to propagate asynchronous errors and how you can keep your code clean while focusing on the happy path. You do this via the use of `map`, `mapError`, and `flatMap`.

Sooner or later you'll use regular throwing functions again to transform your asynchronous data. You'll see how to mix the two error handling idioms by working with throwing functions in combination with `Result`.

After building a solid API, you'll look at a unique `AnyError` type that Apple also offers in combination with `Result`. This type gives you the option to store multiple types of errors inside a `Result`. The benefit is that you can loosen up the error handling strictness without needing to look back to Objective-C by using `NSError`. You'll try out plenty of convenience functions to keep the code concise.

You'll then take a look at the `Never` type to indicate that your code can never fail or succeed. It's a little theoretical but a nice finisher. Consider it a bonus section.

By the end of the chapter, you'll feel comfortable applying powerful transformations to your asynchronous code while dealing with all the errors that can come with it. You'll also be able to avoid the dreaded pyramid of doom and focus on the happy path. But the significant benefit is that your code will be safe and succinct while elegantly handling errors—so let's begin!

11.1 Why use the Result type?

JOIN ME! It's more educational and fun if you can check out the code and follow along with the chapter. You can download the source code at http://mng.bz/5YP1.

Swift's error handling mechanism doesn't translate well to asynchronous error handling. At the time of writing, Swift's asynchronous error handling is still not fleshed out. Generally speaking, developers tend to use Cocoa's style of error handling—coming from the good ol' Objective-C days—where a network call returns multiple values. For instance, you could fetch some JSON data from an API, and the callback gives you both a value *and* an error where you'd have to check for nil on both of them.

Unfortunately, the Cocoa Touch way has some problems—which you'll uncover in a moment—and the `Result` type solves them. The `Result` type, inspired by Rust's `Result` type and the `Either` type in Haskell and Scala, is a functional programming idiom that has been taken on by the Swift community, making it a non-official standard of error handling.

At the time of writing, developers repeatedly reimagine the `Result` type because no official standard exists yet. Even though Swift doesn't officially offer the `Result` type, the Swift Package Manager offers it unofficially. So Apple (indirectly) offers a `Result` type, which justifies implementing it in your codebases. You'll power up `Result` with useful custom functionality as well.

11.1.1 Getting your hands on Result

You can find the `Result` type inside this chapter's playgrounds file. But you can also directly pluck it from the Swift Package Manager—also known as SwiftPM—on GitHub found at http://mng.bz/6GPD.

You can also retrieve `Result` via dependencies of the SwiftPM. This chapter doesn't provide a full guide on how to create a Swift command-line tool via the SwiftPM, but these following commands should get you started.

First, run the following to set up a folder and a Swift executable project. Open the command line and enter the following:

```
mkdir ResultFun
cd ResultFun
swift package init --type executable
```

Next, open Package.swift and change it to the following:

```
// swift-tools-version:4.2
// The swift-tools-
     version declares the minimum version of Swift the required to build this
     package.

import PackageDescription

let package = Package(
    name: "ResultFun",
    dependencies: [
        .package(url: "https://github.com/apple/swift-package-manager",
     from: "0.2.1")
     ],
```

You link to the SwiftPM project from the SwiftPM itself.

```
    targets: [
        .target(
            name: "ResultFun",
            dependencies: ["Utility"]),
     ]
)
```

You need to depend on the Utility package to get required source files.

Inside your project folder, open `Sources/ResultFun/main.swift` and change it to the following:

```
import Basic

let result = Result<String, AnyError>("It's working, hooray!")
print(result)
```

The Basic package is offered by the SwiftPM.

AnyError is covered later in this chapter.

Create a Result type to make sure the import worked correctly.

Type `swift run`, and you'll see `Result(It's working, hooray!)`. Ready? Let's continue.

11.1.2 Result is like Optional, with a twist

`Result` is a *lot* like `Optional`, which is great because if you're comfortable with optionals (see chapter 4), you'll feel right at home with the `Result` type.

Swift's `Result` type is an enum with two cases: namely, a success case and a failure case. But don't let that fool you. `Optional` is also "just" an enum with two cases, but it's powerful, and so is `Result`.

In its simplest form, the `Result` type looks as follows.

Listing 11.1 The `Result` type

```
public enum Result<Value, ErrorType: Swift.Error> {
     /// Indicates success with value in the associated object.
    case success(Value)

    /// Indicates failure with error inside the associated object.
    case failure(ErrorType)

    // ... The rest is left out for later
}
```

The Result type requires two generic values.

In the success case, a Value is bound.

The ErrorType generic is bound in the failure case.

The difference with `Optional` is that instead of a value being present (`some` case) or nil (`none` case), `Result` states that it either has a value (`success` case) or it has an error (`failure` case). In essence, the `Result` type indicates possible failure instead of nil. In other words, with `Result` you can give context for why an operation failed, instead of missing a value.

`Result` contains a value for each case, whereas with `Optional`, only the `some` case has a value. Also the `ErrorType` generic is constrained to Swift's `Error` protocol, which means that only `Error` types can fit inside the `failure` case of `Result`. The constraint comes in handy for some convenience functions, which you'll discover in a later section. Note that the `success` case can fit any type because it isn't constrained.

You haven't seen the full `Result` type, which has plenty of methods, but this code is enough to get you started. Soon enough you'll get to see more methods, such as bridging to and from throwing functions and transforming values and errors inside `Result` in an immutable way.

Let's quickly move on to the *raison d'être* of `Result`: error handling.

11.1.3 Understanding the benefits of Result

To better understand the benefits of the `Result` type in asynchronous calls, let's first look at the downsides of Cocoa Touch–style asynchronous APIs before you see how `Result` is an improvement. Throughout the chapter, you'll keep updating this API with improvements.

Let's look at `URLSession` inside the `Foundation` framework. You'll use `URLSession` to perform a network call, as shown in listing 11.2, and you're interested in the data and error of the response. The iTunes app isn't known for its "popular" desktop application, so you'll create an API for searching the iTunes Store without a desktop app.

To start, you'll use a hardcoded string to search for "iron man"—which you percent encode manually at first—and make use of a function `callURL` to perform a network call.

Listing 11.2 Performing a network call

The @escaping keyword is required in this situation; it indicates that the completionHandler closure can potentially be stored and retain memory.

The callURL function has a completionHandler handler that is called when the URLSession.dataTask finishes.

```
func callURL(with url: URL, completionHandler: @escaping (Data?, Error?)
➥ -> Void) {
    let task = URLSession.shared.dataTask(with: url, completionHandler:
➥ { (data, response, error) -> Void in
        completionHandler(data, error)
    })

    task.resume()
}

let url = URL(string: "https://itunes.apple.com/search?term=iron%20man")!

callURL(with: url) { (data, error) in
    if let error = error {
        print(error)
    } else if let data = data {
        let value = String(data: data, encoding: .utf8)
        print(value)
    } else {
        // What goes here?
    }
}
```

You get the data from a URL, and you pass the data and error back to the caller.

As soon as the callback is called, any error is unwrapped.

You call the callURL function to get the data and error, which are returned at some point in time (asynchronously).

If there is data, you can work with the response.

You turn the data to String to read the raw value.

Here's the problem: If both error and data are nil, what do you do then?

But the problem is that you have to check whether an error and/or the data is nil. Also, what happens if both values are nil? The `URLSession` documentation (http://mng.bz/oVxr) states that either `data` or `error` has a value; yet in code this isn't reflected, and you still have to check against all values.

When returning multiple values from an asynchronous call from `URLSession`, a success and failure value are not mutually exclusive. In theory, you could have received both response data and a failure error or neither. Or you can have one or the other, but falsely assume that if there is no error, the call must have succeeded. Either way, you don't have a compile-time guarantee to enforce safe handling of the returned data. But you're going to change that and see how `Result` will give you these compile-time guarantees.

11.1.4 Creating an API using Result

Let's get back to the API call. With a `Result` type, you can enforce at compile time that a response is either a success (with a value) or a failure (with an error). As an example, let's update the asynchronous call so that it passes a `Result`.

You're going to introduce a `NetworkError` and make the `callURL` function use the `Result` type.

Listing 11.3 A response with `Result`

```
enum NetworkError: Error {
    case fetchFailed(Error)
 }

func callURL(with url: URL, completionHandler: @escaping (Result<Data,
➥ NetworkError>) -> Void) {
    let task = URLSession.shared.dataTask(with: url, completionHandler: {
➥ (data, response, error) -> Void in
      // ... details will be filled in shortly
    })

    task.resume()
}

let url = URL(string: "https://itunes.apple.com/search?term=iron%20man")!

callURL(with: url) { (result: Result<Data, NetworkError>) in
    switch result {
    case .success(let data):
         let value = String(data: data, encoding: .utf8)
         print(value)
    case .failure(let error):
         print(error)
    }
}
```

Define a custom error to pass around inside Result. You can store a lower-level error from URLSession inside the fetchFailed case to help with troubleshooting.

This time, callURL passes a Result type containing either a Data or NetworkError.

Call callURL to get the Result back via a closure.

Pattern match on the success case to get the value out of a Result.

Pattern match on the failure case to catch any error.

As you can see, you receive a `Result<Data, NetworkError>` type when you call `callURL()`. But this time, instead of matching on both `error` and `data`, the values are now mutually exclusive. If you want the value out of `Result`, you *must* handle both cases, giving you compile-time safety in return and removing any awkward situations where both `data` and `error` can be nil or filled at the same time. Also, a big benefit is that you know beforehand that the error inside the `failure` case is of type `NetworkError`, as opposed to throwing functions where you only know the error type at runtime.

You may also use an error handling system where a data type contains an `onSuccess` or `onFailure` closure. But I want to emphasize that with `Result`, if you want the value out, you *must* do something with the error.

AVOIDING ERROR HANDLING

Granted, you can't fully enforce handling an error inside of `Result` if you match on a single case of an enum with the `if case let` statement. Alternatively, you can ignore the error with the infamous `// TODO handle error` comment, but then you'd be consciously going out of your way to avoid handling an error. Generally speaking, if you want to get the value out of `Result`, the compiler tells you to handle the error, too.

As another option, if you're not interested in the reason for failure, yet still want a value out of `Result`, you can get the value out by using the `dematerialize` method. This function either returns the value or throws the error inside `Result`. If you use the `try?` keyword, as shown in the following listing, you can instantly convert the `Result` to an `Optional`.

Listing 11.4 Dematerializing `Result`

```
let value: Data? = try? result.dematerialize()
```

11.1.5 Bridging from Cocoa Touch to Result

Moving on, the response from `URLSession`'s `dataTask` returns three values: `data`, `response`, and `error`.

Listing 11.5 The `URLSession`'s response

```
URLSession.shared.dataTask(with: url, completionHandler: { (data, response,
     error) -> Void in ... }
```

But if you want to work with `Result`, you'll have to convert the values from `URLSession`'s completion handler to a `Result` yourself. Let's take this opportunity to flesh out the `callURL` function so that you can bridge Cocoa Touch–style error handling to a `Result`-style error handling.

One way to convert a value and error to `Result` is to add a custom initializer to `Result` that performs the conversion for you, as shown in the next listing. You can pass this initializer the data and error, and then use that to make a new `Result`. In your `callURL` function, you can then return a `Result` via the closure.

Listing 11.6 Converting a response and error into a `Result`

```
public enum Result<Value, ErrorType> {
      // ... snip

    init(value: Value?, error: ErrorType?) {
        if let error = error {
            self = .failure(error)
        } else if let value = value {
            self = .success(value)
        } else {
            fatalError("Could not create Result")
        }
    }
}

func callURL(with url: URL, completionHandler: @escaping (Result<Data,
➥ NetworkError>) -> Void) {
    let task = URLSession.shared.dataTask(with: url, completionHandler:
➥ { (data, response, error) -> Void in
        let dataTaskError = error.map { NetworkError.fetchFailed($0)}
        let result = Result<Data, NetworkError>(value: data, error:
➥ dataTaskError)
        completionHandler(result)
    })

    task.resume()
}
```

Create an initializer that accepts an optional value and optional error.

If both a value and error are nil, you end up in a bad state and crash, because you can be confident that URLSession returns either a value or error.

Create a Result from the data and error values.

Turn the current error into a higher-level NetworkError and pass the lower-level error from URLSession to its fetchFailed case to help with troubleshooting.

Pass the Result back to the completionHandler closure.

IF AN API DOESN'T RETURN A VALUE Not all APIs return a value, but you can still use `Result` with a so-called *unit type* represented by `Void` or `()`. You can use `Void` or `()` as the value for a `Result`, such as `Result<(), MyError>`.

11.2 Propagating Result

Let's make your API a bit higher-level so that instead of manually creating URLs, you can search for items in the iTunes Store by passing strings. Also, instead of dealing with lower-level errors, let's work with a higher-level `SearchResultError`, which better matches the new search abstraction you're creating. This section is a good opportunity to see how you can propagate and transform any `Result` types.

The API that you'll create allows you to enter a search term, and you'll get JSON results back in the shape of `[String: Any]`.

Listing 11.7 Calling the search API

```
enum SearchResultError: Error {
     case invalidTerm(String)
     case underlyingError(NetworkError)
     case invalidData
 }
```

The invalidTerm case is used when an URL can't be created.

The underlyingError case carries the lower-level NetworkError for troubleshooting or to help recover from an error.

The invalidData case is for when the raw data could not be parsed to JSON.

```
search(term: "Iron man") { result: Result<[String: Any], SearchResultError> in
    print(result)
}
```

Search for a term, and you retrieve a Result via a closure.

11.2.1 Typealiasing for convenience

Before creating the `search` implementation, you create a few typealiases for convenience, which come in handy when repeatedly working with the same `Result` over and over again.

For instance, if you work with many functions that return a `Result<Value, SearchResultError>`, you can define a typealias for the `Result` containing a `SearchResultError`. This typealias is to make sure that `Result` requires only a single generic instead of two by pinning the error generic.

Listing 11.8 Creating a typealias

A generic typealias is defined that pins the Error generic to SearchResultError.

```
typealias SearchResult<Value> = Result<Value, SearchResultError>

let searchResult = SearchResult("Tony Stark")
print(searchResult) // success("Tony Stark")
```

Result offers a convenience initializer to create a Result from a value, if it can deduce the error.

PARTIAL TYPEALIAS The typealias still has a `Value` generic for `Result`, which means that the defined `SearchResult` is pinned to `SearchResultError`, but its value could be anything, such as a `[String: Any]`, `Int`, and so on.

You can create this `SearchResult` by only passing it a value. But its true type is `Result<Value, SearchResultError>`.

Another typealias you can introduce is for the JSON type, namely a dictionary of type `[String: Any]`. This second typealias helps you to make your code more readable, so that you work with `SearchResult<JSON>` in place of the verbose `SearchResult<[String: Any]>` type.

Listing 11.9 The JSON typealias

```
typealias JSON = [String: Any]
```

With these two typealiases in place, you'll be working with the `SearchResult<JSON>` type.

11.2.2 The search function

The new `search` function makes use of the `callURL` function, but it performs two extra tasks: it parses the data to JSON, and it translates the lower-level `NetworkError` to a `SearchResultError`, which makes the function a bit more high-level to use, as shown in the following listing.

Listing 11.10 The `search` function implementation

The function makes use of the JSON and SearchResult typealiases.

The function transforms the search term into a URL-encoded format. Note that encodedString is an optional.

```
func search(term: String, completionHandler: @escaping (SearchResult<JSON>)
     -> Void) {
     let encodedString = term.addingPercentEncoding(withAllowedCharacters:
     .urlHostAllowed)
     let path = encodedString.map { "https://itunes.apple.com/search?term="
     + $0 }

   guard let url = path.flatMap(URL.init) else {
        completionHandler(SearchResult(.invalidTerm(term)))
        return
   }

   callURL(with: url) { result in
       switch result {
       case .success(let data):
            if
               let json = try? JSONSerialization.jsonObject(with: data,
    options: []),
               let jsonDictionary = json as? JSON {
               let result = SearchResult<JSON>(jsonDictionary)
               completionHandler(result)
            } else {
               let result = SearchResult<JSON>(.invalidData)
               completionHandler(result)
            }
       case .failure(let error):
            let result = SearchResult<JSON>(.underlyingError(error))
            completionHandler(result)
       }
   }
}
```

Append the encoded string to the iTunes API path. You use map for this to delay unwrapping.

Transform the complete path into a URL via a flatMap. The guard performs the unwrapping action.

You make sure that an URL is created; on failure, you short-circuit the function by calling the closure early.

The original callURL is called to get raw data.

On the success case, the function tries to convert the data to a JSON format [String: Any].

If the data successfully converts to JSON, you can pass it to the completion handler.

On failure of callURL, you translate the lower-level NetworkError to a higher-level SearchResultError, passing the original NetworkError to a SearchResultError for troubleshooting.

If conversion to JSON format fails, you pass a SearchResultError, wrapped in a Result. You can omit SearchResultError because Swift can infer the error type for you.

Thanks to the `search` function, you end up with a higher-level function to search the iTunes API. But, it's still a little bit clunky because you're manually creating multiple result types and calling the `completionHandler` in multiple places. It's quite the boilerplate, and you could possibly forget to call the `completionHandler` in larger functions. Let's clean that up with `map`, `mapError`, and `flatMap` so that you'll transform and propagate a single `Result` type and you'll only need to call `completionHandler` once.

11.3 Transforming values inside Result

Similar to how you can weave optionals through an application and `map` over them (which delays the unwrapping), you can also weave a `Result` through your functions and methods while programming the happy path of your application. In essence, after you obtained a `Result`, you can pass it around, transform it, and only switch on it when you'd like to extract its value or handle its error.

One way to transform a `Result` is via `map`, similar to mapping over `Optional`. Remember how you could `map` over an optional and transform its inner value if present? Same with `Result`: you transform its `success` value if present. Via mapping, in this case, you'd turn `Result<Data, NetworkError>` into `Result<JSON, NetworkError>`.

Related to how `map` ignores nil values on optionals, `map` also ignores errors on `Result` (see figure 11.1).

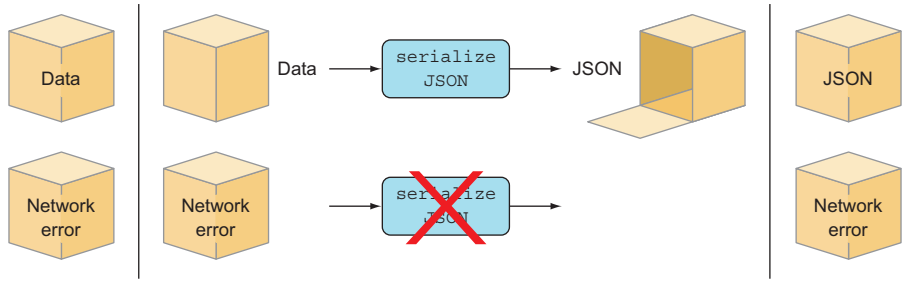


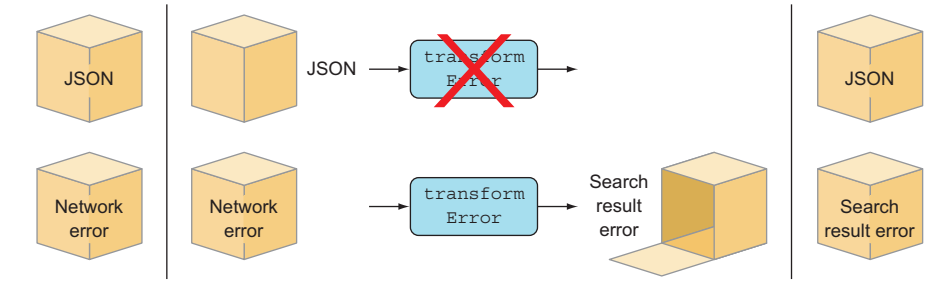
Figure 11.1 Mapping over a `Result`

As a special addition, you can *also* `map` over an error instead of a value inside `Result`. Having `mapError` is convenient because you translate a `NetworkError` inside `Result` to a `SearchResultError`.

With `mapError`, you'd therefore turn `Result<JSON, NetworkError>` into `Result<JSON, SearchResultError>`, which matches the type you pass to the `completionHandler` (see figure 11.2).

With the power of both `map` and `mapError` combined, you can turn a `Result<Data, NetworkError>` into a `Result<JSON, SearchResultError>`, aka `SearchResult<JSON>`, without having to switch on a result once (see figure 11.3). The listing 11.11 gives an example of mapping over an error and value.

Applying `mapError` and `map` help you remove some boilerplate from earlier in the `search` function.



1. You have two results: one with a value and one with an error.
2. mapError does nothing with a successful result.

   With mapError, you apply a function to the error inside a result.
3. The successful result is still the same.

   mapError rewraps the transformed error in a result.

Figure 11.2 Mapping over an error inside `Result`

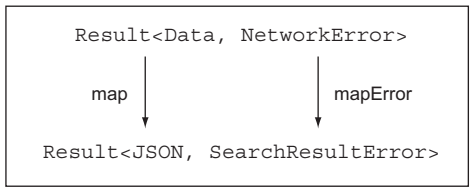


Figure 11.3 Mapping over both the value and error

Listing 11.11 Mapping over an error and value

```
func search(term: String, completionHandler: @escaping (SearchResult<JSON>)
➥ -> Void) {
    // ... snip

    callURL(with: url) { result in

        let convertedResult: SearchResult<JSON> =
            result
                // Transform Data to JSON
                .map { (data: Data) -> JSON in
                    guard
                        let json = try? JSONSerialization.jsonObject(with:
➥ data, options: []),
                        let jsonDictionary = json as? JSON else {
                            return [:]
                        }

                    return jsonDictionary
                }
                // Transform NetworkError to SearchResultError
                .mapError { (networkError: NetworkError) ->
➥ SearchResultError in
```

This result is of type Result<Data, NetworkError>.

On success, you map the data to a JSON.

On failure, you now end up with an empty JSON instead of an error, which you'll solve with flatMap in a moment.

You map the error so that the error type matches SearchResultError.

```
                    return SearchResultError.underlyingError(networkError)
➥ // Handle error from lower layer
        }

        completionHandler(convertedResult)   <-- You pass the SearchResult<JSON>
    }                                            type to the completionHandler after
}                                                all is done.
```

Now, instead of manually unwrapping result types and passing them to the `completionHandler` in multiple flows, you transform the `Result` to a `SearchResult`, and pass it to the `completionHandler` only once. Just like with optionals, you delay any error handling until you want to get the value out.

Unfortunately, `mapError` is not part of the `Result` type offered by Apple. You have to define the method yourself (see the upcoming exercise), but you can also look inside the relevant playgrounds file.

As the next step for improvement, let's improve failure, because currently you're returning an empty dictionary instead of throwing an error. You'll improve this with `flatMap`.

11.3.1 Exercise

1 By looking at the `map` function on `Result`, see if you can create `mapError`.

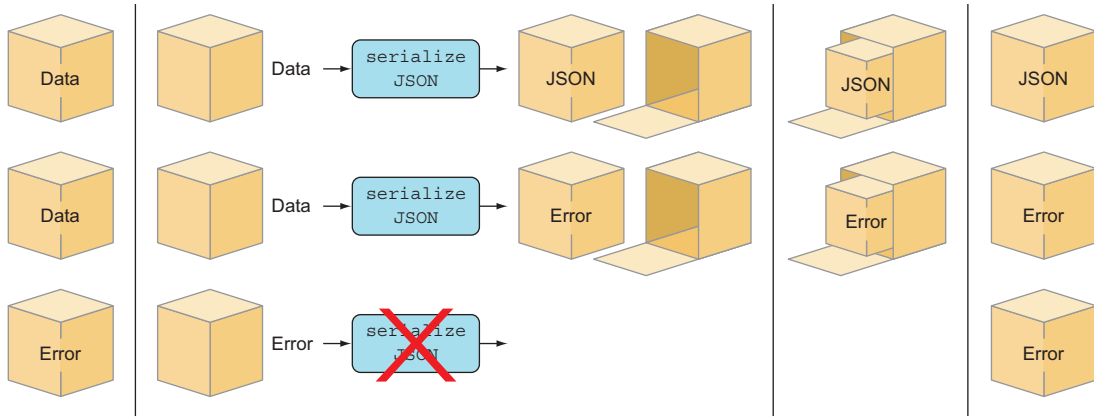
11.3.2 flatMapping over Result

One missing piece from your search function is that when the data can't be converted to JSON format, you'd need to obtain an error. You could throw, but throwing is somewhat awkward because you would be mixing Swift's throwing idiom with the `Result` idiom. You'll take a look at that in the next section.

To stay in the `Result` way of thinking, let's return another `Result` from inside `map`. But you may have guessed that returning a `Result` from a mapping operation leaves you with a nested `Result`, such as `SearchResult<SearchResult<JSON>>`. You can make use of `flatMap`—that is defined on `Result`—to get rid of one extra layer of nesting.

Exactly like how you can use `flatMap` to turn `Optional<Optional<JSON>>` into `Optional<JSON>`, you can also turn `SearchResult<SearchResult<JSON>>` into `SearchResult<JSON>` (see figure 11.4).

By replacing `map` with `flatMap` when parsing `Data` to `JSON`, you can return an error `Result` from inside the `flatMap` operation when parsing fails, as shown in listing 11.12.



1. You start with a sucessful result containing Data (x2) and with one result containing an error.
2. With flatMap, you apply a function to the value inside the result. This function will itself return a new result. This new result could be successful and carry a value, or be a failure result containing an error.

   But if you start with a result containing an error, any flatMap action is ignored.
3. You end up with a nested result.

   If you start with an error, then nothing is transformed or nested.
4. The nested result is flattened to a regular result.

   If you start with an error, nothing happened and the result remains the same.

Figure 11.4 How `flatMap` works on `Result`

Listing 11.12 `flatMaping` over `Result`

```
func search(term: String, completionHandler: @escaping (SearchResult<JSON>)
➥ -> Void) {
      // ... snip

    callURL(with: url) { result in

        let convertedResult: SearchResult<JSON> =
            result
                // Transform error type to SearchResultError
                .mapError { (networkError: NetworkError) ->
➥ SearchResultError in
                    return SearchResultError.underlyingError(networkError)
                }
                // Parse Data to JSON, or return SearchResultError
                .flatMap { (data: Data) -> SearchResult<JSON> in
                    guard
                        let json = try? JSONSerialization.jsonObject(with:
➥ data, options: []),
                        let jsonDictionary = json as? JSON else {
                            return SearchResult(.invalidData)
                    }

                    return SearchResult(jsonDictionary)
        }
```

mapError is moved higher up the chain, so that the error type is SearchResultError before you flatMap over the value. This helps the flatMap so that it can also return SearchResultError instead of NetworkError.

The map operation is replaced by flatMap.

Now you can return a Result from inside a flatMap operation.

```
            completionHandler(convertedResult)
        }
    }
```

FLATMAP DOESN'T CHANGE THE ERROR TYPE A `flatMap` operation on `Result` doesn't change an error type from one to another. For instance, you can't turn `Result<Value, SearchResultError>` to a `Result<Value, NetworkError>` via a `flatMap` operation. This is something to keep in mind and why `mapError` is moved up the chain.

11.3.3 Exercises

2 Using the techniques you've learned, try to connect to a real API. See if you can implement the FourSquare API (http://mng.bz/nxVg) and obtain the venues JSON. You can register to receive free developer credentials.

Be sure to use `Result` to return any venues that you can get from the API.

To allow for asynchronous calls inside playgrounds, add the following:

```
import PlaygroundSupport
PlaygroundPage.current.needsIndefiniteExecution = true
```

3 See if you can use `map`, `mapError`, and even `flatMap` to transform the result so that you call the completion handler only once.

4 The server can return an error, even if the call succeeds. For example, if you pass a latitude and longitude of 0, you get an `errorType` and `errorDetail` value in the `meta` key in the JSON, like so:

```
{"meta":{"code":400,"errorType":"param_error","errorDetail":"Must
➥ provide parameters (ll and radius) or (sw and ne) or (near and
➥ radius)","requestId":"5a9c09ba9fb6b70cfe3f2e12"},"response":{}}
```

Try to make sure that this error is reflected in the `Result` type.

11.4 Mixing Result with throwing functions

Earlier, you avoided throwing an error inside a `Result`'s mapping or flatmapping operation so that you could focus on one idiom at a time.

Let's up the ante. Once you start working with returned data, you'll most likely be using synchronous "regular" functions for processing data, such as parsing or storing data or validating values. In other words, you'll be applying throwing functions to a value inside `Result`. In essence, you're mixing two idioms of error handling.

11.4.1 From throwing to a Result type

Previously, you were parsing data to `JSON` from inside the `flatMap` operation. To mimic a real-world scenario, let's rewrite the `flatMap` operation so that this time you'll be converting `Data` to `JSON` using a throwing function called `parseData`. To make it more realistic, `parseData` comes with an error called `ParsingError`, which deviates from the `SearchResultError` you've been using.

Listing 11.13 The `parseData` function

```
enum ParsingError: Error {
     case couldNotParseJSON
}

func parseData(_ data: Data) throws -> JSON {
    guard
        let json = try? JSONSerialization.jsonObject(with: data, options: []),
        let jsonDictionary = json as? JSON else {
            throw ParsingError.couldNotParseJSON
    }
    return jsonDictionary
}
```

A specific error used for parsing data

The parseData function turns Data into JSON and can throw a ParsingError.

You can turn this throwing function into a `Result` via an initializer on `Result`. The initializer accepts a closure that may throw; then the `Result` initializer catches any errors thrown from the closure and creates a `Result` out of it. This `Result` can be successful or failing (if an error has been thrown).

It works as follows: you pass a throwing function to `Result` and, in this case, have it convert to `Result<JSON, SearchResultError>`.

Listing 11.14 Converting a throwing function to `Result`

```
let searchResult: Result<JSON, SearchResultError> = Result(try parseData(data))
```

You're almost there, but one thing is missing. You try to convert `parseData` to a `Result` with a `SearchResultError` via an initializer. Yet, `parseData` doesn't throw a `SearchResultError`. You can look in the body of `parseData` to confirm. But Swift only knows at runtime what error `parseData` throws.

If during conversion any error slips out that is not `SearchResultError`, the initializer on `Result` throws the error from `parseData`, which means that you need to catch that error, too. Moreover, this is why the initializer on `Result` is throwing, because it throws any errors that it can't convert. This awkwardness is a bit of the pain you have when turning a runtime-known error into a compile-time-known error.

To complete the conversion, you need to add a *do catch* statement; you remain in the `do` block on success or when `Result` receives a `SearchResultError`. But as soon as `parseData` throws a `ParsingError`, as shown in the following example, you end up in the `catch` block, which is an opportunity to fall back to a default error.

Listing 11.15 Passing a throwing function to `Result`

```
do {
  let searchResult: Result<JSON, SearchResultError> = Result(try
parseData(data))
 } catch {
  print(error) // ParsingError.couldNotParseData
```

You call parseData(); if it succeeds you have a searchResult.

```
    let searchResult: Result<JSON, SearchResultError> =
        Result(.invalidData(data))
}
```

If conversion fails, you end up in the catch statement, where you default back to returning a SearchResult with default error.

11.4.2 Converting a throwing function inside flatMap

Now that you know how to convert a throwing function to `Result`, you can start mixing these in with your pipeline via `flatMap`.

Inside the `flatMap` method from earlier, create a `Result` from the throwing `parseData` function.

Listing 11.16 Creating a `Result` from `parseData`

```
func search(term: String, completionHandler: @escaping (SearchResult<JSON>)
➥ -> Void) {
    // ... snip

    callURL(with: url) { result in
        let convertedResult: SearchResult<JSON> =
            result
                .mapError { SearchResultError.underlyingError($0) }
                .flatMap { (data: Data) -> SearchResult<JSON> in
                    do {
                        // Catch if the parseData method throws a ParsingError.
                        let searchResult: SearchResultError<JSON> =
➥ Result(try parseData(data))
                        return searchResult
                    } catch {
                        // You ignore any errors that parseData throws and
➥ revert to SearchResultError.
                        return SearchResult(.invalidData(data))
                    }
        }

        completionHandler(convertedResult)
    }
}
```

You're entering a flatMap operation.

The parseData function is passed to the initializer.

If the parseData conversion fails, you end up in the catch statement and default back to SearchResultError.invalidData.

11.4.3 Weaving errors through a pipeline

By composing `Result` with functions via mapping and flatmapping, you're performing so-called *monadic* error handling. Don't let the term scare you—`flatMap` is based on *monad* laws from functional programming. The beauty is that you can focus on the happy path of transforming your data.

As with optionals, `flatMap` isn't called if `Result` doesn't contain a value. You can work with the real value (whether `Result` is erroneous or not) while carrying an error context and propagate the `Result` higher—all the way to where some code can pattern match on it, such as the caller of a function.

As an example, if you were to continue the data transformations, you could end up with multiple chained operations. In this pipeline, `map` would always keep you on the

happy path, and with `flatMap` you could short-circuit and move to either the happy path or error path.

For instance, let's say you want to add more steps, such as validating data, filtering it, and storing it inside a database (perhaps a cache). You would have multiple steps where `flatMap` could take you to an error path. In contrast, `map` always keeps you on the happy path (see figure 11.5).

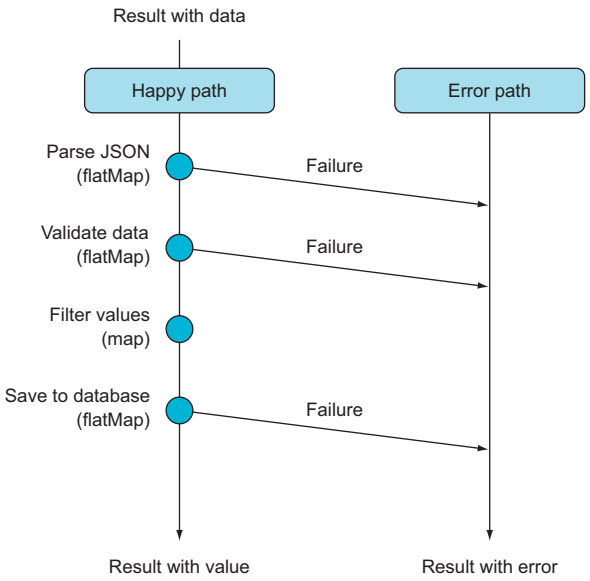


Figure 11.5 Happy path programming

For the sake of brevity, you aren't going to implement all these methods, but the point is that you can build a sophisticated pipeline, as shown in the following listing, weave the error through it, and only call the completion handler once.

Listing 11.17 A longer pipeline

```
func search(term: String, completionHandler: @escaping (SearchResult<JSON>)
     -> Void) {
      // ... snip

    callURL(with: url) { result in

        let convertedResult: SearchResult<JSON> =
            result
                // Transform error type to SearchResultError
                .mapError { (networkError: NetworkError) ->
     SearchResultError in
                  // code omitted
                }
                // Parse Data to JSON, or return SearchResultError
                .flatMap { (data: Data) -> SearchResult<JSON> in
                  // code omitted
                }
```

```
                // validate Data
                .flatMap { (json: JSON) -> SearchResult<JSON> in
                  // code omitted
                }
                // filter values
                .map { (json: JSON) -> [JSON] in
                  // code omitted
                }
                // Save to database
                .flatMap { (mediaItems: [JSON]) -> SearchResult<JSON> in
                  // code omitted
                  database.store(mediaItems)
            }

            completionHandler(convertedResult)
        }
    }
```

SHORT-CIRCUITING A CHAINING OPERATION Note that `map` and `flatMap` are ignored if `Result` contains an error. If any `flatMap` operation returns a `Result` containing an error, any subsequent `flatMap` and `map` operations are ignored as well.

With `flatMap` you can short-circuit operations, just like with `flatMap` on `Optional`.

11.4.4 Finishing up

It may not look like much, but your API packs quite the punch. It handles network errors and parsing errors, and it's easy to read and to extend. And still you avoid having an ugly pyramid of doom, and your code focuses on the happy path. On top of that, calling `search` means that you only need to switch on the `Result`.

Receiving a simple `Result` enum looks a little underwhelming after all that work. But clean APIs tend to appear simple from time to time.

11.4.5 Exercise

5 Given the following throwing functions, see if you can use them to transform `Result` in your FourSquare API:

```
func parseData(_ data: Data) throws -> JSON {
    guard
        let json = try? JSONSerialization.jsonObject(with: data,
  options: []),
        let jsonDictionary = json as? JSON else {
            throw FourSquareError.couldNotParseData
    }
    return jsonDictionary
}

func validateResponse(json: JSON) throws -> JSON {
    if
        let meta = json["meta"] as? JSON,
```

```
            let errorType = meta["errorType"] as? String,
            let errorDetail = meta["errorDetail"] as? String {
            throw FourSquareError.serverError(errorType: errorType,
      errorDetail: errorDetail)
        }

        return json
    }

    func extractVenues(json:  JSON) throws -> [JSON] {
        guard
            let response = json["response"] as? JSON,
            let venues = response["venues"] as? [JSON]
            else {
                throw FourSquareError.couldNotParseData
        }
        return venues
    }
```

11.5 Multiple errors inside of Result

Working with `Result` may feel constricting at times when multiple actions can fail. Previously, you were translating each failure into a `Result` holding a single error type—`SearchResultError` in the examples. Translating errors to a single error type is a good practice to follow. But it may get burdensome moving forward if you're dealing with many different errors, especially when you're beginning a new project and you need to glue together all kinds of throwing methods. Translating every error to the correct type may slow you down.

Not to worry; if you want to move fast and keep errors known at runtime, you can use a generic type called `AnyError`—also offered by the Swift Package Manager.

11.5.1 Introducing AnyError

`AnyError` represents any error that could be inside `Result`, allowing you to mix and match all types of errors in the same `Result` type. With `AnyError`, you avoid having to figure out each error at compile time.

`AnyError` wraps around an `Error` and stores the error inside; then a `Result` can have `AnyError` as its error type, such as `Result<String, AnyError>`. You can manually create an `AnyError`, but you can also create a `Result` of type `Result<String, AnyError>` in multiple ways.

Notice how `Result` has two initializers specialized to `AnyError`: one converts a regular error to `AnyError`, the other accepts a throwing function in which the error converts to `AnyError`.

Listing 11.18 Creating a `Result` with `AnyError`

```
enum PaymentError: Error {
    case amountTooLow
    case insufficientFunds
}
```

```
let error: AnyError = AnyError(PaymentError.amountTooLow)

let result: Result<String, AnyError> = Result(PaymentError.amountTooLow)

let otherResult: Result<String, AnyError> = Result(anyError: { () throws ->
     String in
     throw PaymentError.insufficientFunds
})
```

You can pass an error to the AnyError type yourself.

You can also pass an error to Result directly, which automatically converts an error to AnyError because the Result is of type Result<String, AnyError>.

You can even pass throwing functions to Result; because AnyError represents all possible errors, the conversion always succeeds.

Functions returning a `Result` with `AnyError` are similar to a throwing function where you only know the error type at runtime.

Having `AnyError` makes sense when you're developing an API and don't want to focus too much on the proper errors yet. Imagine that you're creating a function to transfer money, called `processPayment`. You can return different types of errors in each step, which relieves you of the burden of translating different errors to one specific type. Notice how you also get a special `mapAny` method.

Listing 11.19 Returning different errors

```
func processPayment(fromAccount: Account, toAccount: Account, amountInCents:
➥ Int, completion: @escaping (Result<String, AnyError>) -> Void) {

    guard amountInCents > 0 else {
         completion(Result(PaymentError.amountTooLow))
         return
    }

    guard isValid(toAccount) && isValid(fromAccount) else {
         completion(Result(AccountError.invalidAccount))
         return
    }

    // Process payment

    moneyAPI.transfer(amountInCents, from: fromAccount, to: toAccount) {
➥ (result: Result<Data, AnyError>) in
         let response = result.mapAny(parseResponse)
         completion(response)
    }

}
```

Return a PaymentError here.

But you can return a different error here, of type AccountError.

Utilize a special mapAny method.

An interesting thing to note is that if `Result` has `AnyError` as its type, you gain a special `mapAny` method for free. The `mapAny` method works similarly to `map`, except that it can accept any throwing function. If a function inside `mapAny` throws, `mapAny` automatically wraps this error inside an `AnyError`. This technique allows you to pass throwing functions to `map` without requiring you to catch any errors.

Also, a big difference with `flatMap` is that you could not change the `ErrorType` from within the operations. With `flatMap`, you would have to create and return a new `Result` manually. With `mapAny`, you can pass a regular throwing function and let `mapAny` handle the catching and wrapping into `AnyError`. Applying `mapAny` allows you to map over the value and even change the error inside `Result`.

> **HOW TO CHOOSE BETWEEN MAP OR MAPANY** The difference between `map` and `mapAny` is that `map` works on all `Result` types, but it doesn't catch errors from throwing functions. In contrast, `mapAny` works on both throwing and non-throwing functions, but it's available only on `Result` types containing `AnyError`. Try to use `map` when you can; it communicates that a function cannot throw. Also, if you ever refactor `AnyError` back to a regular `Error` inside `Result`, then `map` is still available.

MATCHING WITH ANYERROR

To get the error out when dealing with `AnyError`, you can use the `underlyingError` property of `AnyError` to match on the actual error inside of it.

Listing 11.20 Matching on `AnyError`

```
processPayment(fromAccount: from, toAccount: to, amountInCents: 100) {
➥ (result: Result<String, AnyError>) in
    switch result {
    case .success(let value): print(value)
    case .failure(let error) where error.underlyingError is AccountError:
        print("Account error")
    case .failure(let error):
        print(error)
    }
}
```

`AnyError` is a useful placeholder to let you handle "proper" error handling at a later time. When time permits and your code solidifies, you can start replacing the general errors with stricter error translations for extra compile-time benefits.

Working with `AnyError` gives you a lot more flexibility. But you suffer somewhat from code erosion because you lose a big benefit of `Result`, which is being able to see which errors you can expect before even running your code. You may also consider `NSError` instead of `AnyError` because `NSError` is also flexible. But then you'll be looking back to Objective-C, and you also lose the benefits of using Swift errors, such as strong pattern matching on enum-type errors. Before going the `NSError` route, you may want to reconsider and see if you get to keep using Swift errors in combination with `AnyError`.

11.6 Impossible failure and Result

Sometimes you may need to conform to a protocol that wants you to use a `Result` type. But the type that implements the protocol may never fail. Let's see how you can improve your code in this scenario with a unique tidbit. This section is a bit esoteric and theoretical, but it proves useful when you run into a similar situation.

11.6.1 When a protocol defines a Result

Imagine that you have a `Service` protocol representing a type that loads some data for you. This `Service` protocol determines that data is to be loaded asynchronously, and it makes use of a `Result`.

You have multiple types of errors and data that can be loaded, so `Service` defines them as associated types.

Listing 11.21 The `Service` protocol

```
protocol Service {
     associatedtype Value
     associatedtype Err: Error
     func load(complete: @escaping (Result<Value, Err>) -> Void)
 }
```

The Value that the Service loads

This is the Error the Service can give. Note how your associated type is called Err, and it's constrained to the Error protocol.

The load method returns a Result containing a Value and Err, passed by a completion closure.

Now you want to implement this `Service` by a type called `SubscriptionsLoader`, which loads a customer's subscriptions for magazines. This is shown in listing 11.22. Note that loading subscriptions *always* succeeds, which you can guarantee because they are loaded from memory. But the `Service` type declares that you use `Result`, which needs an error, so you do need to declare what error a `SubscriptionsLoader` throws. `SubscriptionsLoader` doesn't have errors to throw. To remedy this problem, let's create an empty enum—conforming to `Error`—called `BogusError` so that `SubscriptionsLoader` can conform to `Service` protocol. Notice that `BogusError` has no cases, meaning that nothing can actually create this enum.

Listing 11.22 Implementing the `Service` protocol

```
struct Subscription {
   // ... details omitted
}

enum BogusError: Error {}

final class SubscriptionsLoader: Service {
    func load(complete: @escaping (Result<[Subscription], BogusError>) ->
     Void) {
         // ... load data. Always succeeds
        let subscriptions = [Subscription(), Subscription()]
        complete(Result(subscriptions))
    }
}
```

The Subscription is the type of data retrieved from SubscriptionsLoader.

You create a dummy error type so that you can define it on the Result type, in order to please Service.

The load method now returns a Result returning an array of subscriptions. Notice how you defined the uninhabitable BogusError type to please the protocol.

You made an empty enum that conforms to `Error` merely to please the compiler. But because `BogusError` has no cases, you can't instantiate it, and Swift knows this. Once you call `load` on `SubscriptionsLoader` and retrieve the `Result`, you can match *only*

on the `success` case, and Swift is smart enough to understand that you can *never* have a failure case. To emphasize, a `BogusError` can *never* be created, so you don't need to match on this, as the following example shows.

Listing 11.23 Matching only on the success case

```
let subscriptionsLoader = SubscriptionsLoader()
subscriptionsLoader.load { (result: Result<[Subscription], BogusError>) in
    switch result {
    case .success(let subscriptions): print(subscriptions)
        // You don't need .failure
    }
}
```

Swift lets you get away with this. Normally you'd get a compiler error!

This technique gives you compile-time elimination of cases to match on and can clean up your APIs and show clearer intent. But an official solution—the `Never` type—lets you get rid of `BogusError`.

THE NEVER TYPE

To please the compiler, you made a bogus error type that can't be instantiated. Actually, such a type already exists in Swift and is called the `Never` type.

The `Never` type is a so-called *bottom type*; it tells the compiler that a certain code path can't be reached. You may also find this mechanism in other programming languages, such as the `Nothing` type in Scala, or when a function in Rust returns an exclamation mark (`!`).

`Never` is a hidden type used by Swift to indicate impossible paths. For example, when a function calls a `fatalError`, it can return a `Never` type, indicating that returning something is an impossible path.

Listing 11.24 From the Swift source

```
func crashAndBurn() -> Never {
    fatalError("Something very, very bad happened")
}
```

The Never type is returned, but the code guarantees it never returns.

If you look inside the Swift source, you can see that `Never` is nothing but an empty enum.

Listing 11.25 The `Never` type

```
public enum Never {}
```

In your situation, you can replace your `BogusError` with `Never` and get the same result. You do, however, need to make sure that `Never` implements `Error`.

Listing 11.26 Implementing `Never`

```
extension Never: Error {}

final class SubscriptionsLoader: Service {
    func load(complete: @escaping (Result<[Subscription], Never>) -> Void) {
         // ... load data. Always succeeds
        let subscriptions = [Subscription(), Subscription()]
        complete(Result(subscriptions))
    }
}
```

You extend Never to make it conform to the Error protocol.

You now use the Never type to indicate that your SubscriptionsLoader never fails.

NOTE From Swift 5 on, `Never` conforms to some protocols, like `Error`.

Notice that `Never` can also indicate that a service never succeeds. For instance, you can put the `Never` as the success case of a `Result`.

11.7 Closing thoughts

I hope that you can see the benefits of error handling with `Result`. You've seen how `Result` can give you compile-time insights into which error to expect. Along the way you took your `map` and `flatMap` knowledge and wrote code that pretended to be error-free, yet was carrying an error-context. Now you know how to apply monadic error handling.

Here's a controversial thought: you can use the `Result` type for *all* the error handling in your project. You get more compile-time benefits, but at the price of more difficult programming. Error handling is more rigid with `Result`, but your code will be safer and stricter as a reward. And if you want to speed up your work a little, you can always create a `Result` type containing `AnyError` and take it from there.

Summary

- Using the default way of `URLSession`'s data tasks is an error-prone way of error handling.
- `Result` is offered by the Swift Package Manager and is a good way to handle asynchronous error handling.
- `Result` has two generics and is a lot like `Optional`, but has a context of why something failed.
- `Result` is a compile-time safe way of error handling, and you can see which error to expect before running a program.
- By using `map` and `flatMap` and `mapError`, you can cleanly chain transformations of your data while carrying an error context.
- Throwing functions can be converted to a `Result` via a special throwing initializer. This initializer allows you to mix and match two error throwing idioms.
- You can postpone strict error handling with the use of `AnyError`.
- With `AnyError`, multiple errors can live inside `Result`.
- If you're working with many types of errors, working with `AnyError` can be faster, at the expense of not knowing which errors to expect at compile time.

- `AnyError` can be a good alternative to `NSError` so that you reap the benefits of Swift error types.
- You can use the `Never` type to indicate that a `Result` can't have a `failure` case, or a `success` case.

Answers

1 By looking at the `map` function on `Result`, see if you can create `mapError`:

```
extension Result {

    public func mapError<E: Error>(_ transform: (ErrorType) throws
➥ -> E) rethrows -> Result<Value, E> {
        switch self {
        case .success(let value):
            return Result<Value, E>(value)
        case .failure(let error):
            return Result<Value, E>(try transform(error))
        }
    }

}
```

The following part is the answer to exercises 2 and 3:

2 Using the techniques you've learned, try to connect to a real API. See if you can implement the FourSquare API (http://mng.bz/nxVg) and obtain the venues JSON. You can register to receive free developer credentials.

3 See if you can use `map`, `mapError`, and even `flatMap` to transform the result, so that you call the completion handler only once.

4 The server can return an error, even if the call succeeds. For example, if you pass a latitude and longitude of 0, you get an `errorType` and `errorDetail` value in the `meta` key in the JSON. Try to make sure that this error is reflected in the `Result` type:

```
// You need an error
enum FourSquareError: Error {
    case couldNotCreateURL
    case networkError(Error)
    case serverError(errorType: String, errorDetail: String)
    case couldNotParseData
}

let clientId = ENTER_YOUR_ID
let clientSecret = ENTER_YOUR_SECRET
let apiVersion = "20180403"

// A helper function to create a URL
func createURL(endpoint: String, parameters: [String: String]) -> URL? {
    let baseURL = "https://api.foursquare.com/v2/"
```

```
    // You convert the parameters dictionary in an array of URLQueryItems
    var queryItems = parameters.map { pair -> URLQueryItem in
        return URLQueryItem(name: pair.key, value: pair.value)
    }

    // Add default parameters to query
    queryItems.append(URLQueryItem(name: "v", value: apiVersion))
    queryItems.append(URLQueryItem(name: "client_id", value: clientId))
    queryItems.append(URLQueryItem(name: "client_secret", value:
  clientSecret))

    var components = URLComponents(string: baseURL + endpoint)
    components?.queryItems = queryItems
    return components?.url
}

// The getvenues call
func getVenues(latitude: Double, longitude: Double, completion:
➥ @escaping (Result<[JSON], FourSquareError>) -> Void) {
    let parameters = [
        "ll": "\(latitude),\(longitude)",
        "intent": "browse",
        "radius": "250"
    ]

    guard let url = createURL(endpoint: "venues/search", parameters:
  parameters)
        else {
            completion(Result(.couldNotCreateURL))
            return
    }

    let task = URLSession.shared.dataTask(with: url) { data, response,
  error in
        let translatedError = error.map { FourSquareError.networkError(
  $0) }
        // Convert optional data and optional to Result
        let result = Result<Data, FourSquareError>(value: data, error:
  translatedError)
            // Parsing Data to JSON
            .flatMap { data in
                guard
                    let rawJson = try?
  JSONSerialization.jsonObject(with: data, options: []),
                    let json = rawJson as? JSON
                    else {
                        return Result(.couldNotParseData)
                }
                return Result(json)
            }
            // Check for server errors
            .flatMap { (json: JSON) -> Result<JSON, FourSquareError> in
                if
                    let meta = json["meta"] as? JSON,
                    let errorType = meta["errorType"] as? String,
```

```
                    let errorDetail = meta["errorDetail"] as? String {
                    return Result(.serverError(errorType: errorType,
   errorDetail: errorDetail))
                }

                return Result(json)
            }
            // Extract venues
            .flatMap { (json: JSON) -
   > Result<[JSON], FourSquareError> in
                guard
                    let response = json["response"] as? JSON,
                    let venues = response["venues"] as? [JSON]
                    else {
                        return Result(.couldNotParseData)
                }
                return Result(venues)
        }

        completion(result)
    }

    task.resume()
}

// Times square
let latitude = 40.758896
let longitude = -73.985130

// Calling getVenues

getVenues(latitude: latitude, longitude: longitude) { (result:
   Result<[JSON], FourSquareError>) in
    switch result {
    case .success(let categories): print(categories)
    case .failure(let error): print(error)
    }
}
```

5 Given the throwing functions, see if you can use them to transform `Result` in your FourSquare API:

```
enum FourSquareError: Error {
    // ... snip
    case unexpectedError(Error) // Adding new error for when conversion
   to Result fails
}

func getVenues(latitude: Double, longitude: Double, completion:
➥ @escaping (Result<[JSON], FourSquareError>) -> Void) {
    // ... snip
    let result = Result<Data, FourSquareError>(value: data, error:
   translatedError)
            // Parsing Data to JSON
```

```
        .flatMap { data in
            do {
                return Result(try parseData(data))
            } catch {
                return Result(.unexpectedError(error))
            }
        }
        // Check for server errors
        .flatMap { (json: JSON) -> Result<JSON, FourSquareError> in
            do {
                return Result(try validateResponse(json: json))
            } catch {
                return Result(.unexpectedError(error))
            }
        }
        // Extract venues
        .flatMap { (json: JSON) -> Result<[JSON], FourSquareError> in
            do {
                return Result(try extractVenues(json: json))
            } catch {
                return Result(.unexpectedError(error))
            }
}
```

iOS DEVELOPMENT

Swift IN DEPTH

Tjeerd in 't Veen

It's fun to create your first toy iOS or Mac app in Swift. Writing secure, reliable, professional-grade software is a different animal altogether. The Swift language includes an amazing set of high-powered features, and it supports a wide range of programming styles and techniques. You just have to roll up your sleeves and learn Swift in depth.

Swift in Depth guides you concept by concept through the skills you need to build professional software for Apple platforms, such as iOS and Mac; also on the server with Linux. By following the numerous concrete examples, enlightening explanations, and engaging exercises, you'll finally grok powerful techniques like generics, efficient error handling, protocol-oriented programming, and advanced Swift patterns. Author Tjeerd in 't Veen reveals the high-value, difficult-to-discover Swift techniques he's learned through his own hard-won experience.

What's Inside

- Writing reusable code with generics
- Iterators, sequences, and collections
- Protocol-oriented programming
- Understanding map, flatMap, and compactMap
- Asynchronous error handling with Result
- Best practices in Swift

Written for advanced-beginner and intermediate-level Swift programmers.

Tjeerd in 't Veen is a senior software engineer and architect in the mobile division of a large international banking firm.

"An excellent guide to using the advanced features of Swift to produce clean, high-performing code. The content is masterfully delivered, making it easy to quickly level-up your skills."
—Jason Pike, Atlas RFID Solutions

"Highly recommended to anyone interested in the Apple platform. For the novice who wants to become an expert, this is definitely where you should start!"
—Helmut Reiterer
Revenue Recovery Solutions

"Because Swift is so new, it's hard to find good resources to learn it. Look no further than this book."
—Tyler Slater, Jolt

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/swift-in-depth

ISBN-13: 978-1-61729-518-8
ISBN 10: 1-61729-518-3
54999
9 781617 295188

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]