Functional Programming in Typescript
A Practical Guide

Sam Van Overmeire

Functional Programming in Typescript

A Practical Guide

Sam Van Overmeire

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

This version was published on 2021-10-03

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.

© 2020 - 2021 Sam Van Overmeire

Contents

Acknowledgments **1**
About the book cover 1

Introduction **2**
Caveat Lector 2
Why FP? 2
Why Javascript? 3
Why Typescript? 6
Why FP-TS? 9
Motivation for this book 10
What you should know 10
Overview 10

Tidbits: Some Useful Javascript and Typescript **12**
Requirements 12
Setting up a new project 12
More than one way to build a function 13
Backticks for easy printing 14
The ternary operator 15
Discriminating unions 16
Summary 18

First Steps with Functional Programming **19**
What is Functional Programming 19
Pure functions, referential transparency and immutability 19
Writing functional code 21
Writing functional programs 24
Monads 30
Briefly about types again 34
Summary 36

Our First Project: User Registration **37**
Defining our types 37
Validation with Either 42
Before we continue: Either or Validation? 45

Creating our user with lifting . . . 47
Bringing it together . . . 52
Change is inevitable - how to cope with it . . . 55
Summary . . . 60

Deploying a monitoring application . . . 61
AWS Lambda and SAM . . . 61
Setting up our project . . . 63
Using the Cloudwatch SDK . . . 69
Monads for asynchronous behavior . . . 73
Transforming our data . . . 78
Synchronous behavior and IO . . . 85
Bringing it all together . . . 88
Extensions . . . 93
A small aside: memoization . . . 100
Key points . . . 101
Summary . . . 102

Testing . . . 103
The types of testing . . . 103
You mentioned TDD? . . . 104
FP and testing . . . 105
Property-based testing . . . 116
Other tools for better code . . . 118
Summary . . . 118

Once more unto the breach: our final application . . . 119
Setting up . . . 119
Writing our first Create Lambda code . . . 127
Core logic of the Create Lambda . . . 136
Bringing it together . . . 142
Creating the Retrieve Lambda . . . 146
DynamoDB . . . 156
Adding the real database . . . 158
Summary . . . 167

Expanding our reservation application . . . 169
Using the Reader Monad for gathering config . . . 169
Using State . . . 175
Better error types . . . 180
Restaurants and sequenceS . . . 188
Choosing configs . . . 193
Exploring some more ideas . . . 195
Summary . . . 197

Rewriting an application **198**
The original application 198
Evaluation 211
Improvements overview 215
Our new code setup 216
Validation 226
Dealing with the asynchronous calls 232
Advantages of our functional approach 236
Summary 238

Advanced FP techniques **239**
Domain Specific Languages 239
Tagless final 243
Free monads 243
Executable and declarative encoding, algebraic design 244
Summary 244

Conclusion **246**
What we learned 246
Where to go from here 246

Appendix A: Additional Concepts **248**
Writer 248
These 248

Acknowledgments

Though acknowledgments are probably the least-read part of any book (besides the index and endnotes), they are fraught with danger. So many people who have helped you, influenced you. What if you forget someone? Will they think you did it on purpose? On the other hand, an incomplete thank you is still better than none at all. So here goes.

A big thank you for their love and support to my girlfriend, Annelies Scheppers, my father, Marc Van Overmeire, my mother, Marleen Schamp (whom I miss every day), my brothers, sister, grandparents (especially my grandmother Rafaella Otte), the other members of my family and my friends.

A shout-out to the many colleagues I have had the pleasure of working with throughout the years. I have been very lucky meeting so many outstanding people. And since I met those people on the job, a thank you to the companies who have taken me on board, first XT-I and more recently Branch. Particular thanks to my current boss, Thomas Matthys, for his enthusiastic support for this book.

My thanks also go out to the VRT - where I have worked in the Sporza and User Services teams - for allowing me to draw inspiration from the code I wrote there for one of this book's chapters.

More in general, thanks to the many - often anonymous - people on the Internet who write blogs, create videos or answer questions on Stack Overflow. For almost every problem a programmer faces, bits of useful information can be found on the Internet. All because so many people enjoy helping others.

And finally, thank you reader for picking up this book! Please try to forgive me for whatever faults you find within.

About the book cover

The picture is by Mads Schmidt Rasmussen[1]. It is quite hard to find a fitting image for abstract things like programming. Which is why Manning and O'Reilly put quasi-random pictures of animals and people on their covers. Similarly, I just found a picture that I liked. But if you really want some rationale for the cover, here goes: learning FP can be a challenging climb, but the view will definitely be worth the effort.

[1] https://unsplash.com/photos/xfngap_DToE

Introduction

Where we introduce the subject at hand and explain our choice of language.

Caveat Lector

Which is the smartass way of saying "*Reader, beware*".

I am a couple of things. I'm a professional programmer in the sense that I get paid to sit in front of a screen and write code. Unless I'm in a meeting, in which case I get paid to listen to people while *sneakily* writing code. Most of that code is written in popular languages (Javascript, Java and Python are the top three in my day-to-day), but I have spent a fair amount of time experimenting with a large range of other languages including Go, Haskell, Purescript and Clojure. I am also a writer. Nowadays mostly blog posts, but I have written a book once before.

However, I am not an expert in Functional Programming. My background is in Object Oriented Programming and most of the applications I have been involved with used objects and classes, though several of the more recent ones have turned towards the functional paradigm. So I will try to write this book with the beginner's mind. And perhaps by putting myself on the right path, by forcing myself to think deeper about the concepts of Functional Programming, I will help others along on their path as well.

So do what you should always do when reading a book. Take everything with a grain of salt. Be critical, but with an open mind. When it comes to the more concrete examples in this book, what I offer is often just one way of tackling the problem. And the right solution will depend on context.

Now forward, onto the real introduction.

Why FP?

Somehow, **Functional Programming** (which will be abbreviated as **FP**) feels like the new kid on the block. Which is strange, considering it is a paradigm that slightly predates **Object Oriented Programming** (let us call that **OOP** from now on). This might be a case of 'history is written by the winners'. OOP has been the reigning programming paradigm for several decades now. Outside of the academic world, it is often viewed as the one way all code *should* be written. And it is certainly the way most of it *is* written.

So OOP seems old, familiar and trustworthy. FP is exotic, unknown. At the same time the ideas behind FP are very well suited to our world. It promises a new way to manage the growing

complexity of applications. Yes, OOP promised this as well. But most of us have experienced first-hand that it only partly fulfilled its promise. Spaghetti code is still very real. FP is also keen on helping us deal with the parallelism that other paradigms struggle with. Which was OK decades ago but is troublesome in an age where every computer - and practically every smartphone - has a multicore CPU. Another reason FP may feel fresh and new is because more and more Object Oriented Languages are adding functional 'features'. Java 8 gave us Streams, Java 14 brings records. Other languages have taken a pragmatic approach and let you pick and mix your styles of coding. Scala has been doing this for a long time, Kotlin is a more recent example.

One final reason FP may seem new: no one you know is actually using it yet. You have probably read about its potential and how it will change the way you think about code. But back in the real world, recruiters are asking you about your OOP experience; a large majority of blogs and books talk about OOP; the team you join has applications written in it. Furthermore, the languages used in your company are often well suited for writing code in that paradigm, frustrating your attempts to write beautiful functional code. Take Java and Python. Java is not only very verbose, it does not support functions as **first-class citizens**. Which alone is sufficient to make writing functional code at best a laborious chore. Python's functions are first-class citizens and ceremony is minimal. But even though you can write Python in a functional style, it was not created with the intent of being (primarily) functional. And sometimes that shows.

Somehow I already feel obliged to add a warning: you can write beautiful applications in the OOP paradigm and you can create unmaintainable programs with FP. Furthermore, both styles have their own advantages and disadvantages. However, I think FP has several very useful qualities. Enough to make it a valid choice when starting work on a new application.

Another note: a switch to FP is not a *deus ex machina* for helping a company in a downward spiral get up on its feet again. It is a useful tool, but turning around a company requires changes to its organisation. A change towards self-organizing teams (a radical change for some of the larger command-and-control companies out there), the breaking down of silos, Devops and (real) Agile.

So I picked two of the most popular languages out there (apologies to the Microsoft Stack with which I have no professional experience). What about that other hyper-popular language? Ubiquitous in the frontend and growing in the backend? Javascript.

Why Javascript?

From the title of the book, you may have guessed that I like the possibilities that Javascript offers when it comes to Functional Programming. *Compared to other popular languages like Python or Java*, functional code written in Javascript can be concise and elegant. Yes, Clojure is more concise, Haskell's pattern matching is advanced and its compiler very powerful. But I want to write functional code on the job, and there is a better chance that I will be able to do that with Javascript.

If we want to write functional code in Javascript, we can turn to libraries like Lodash, Ramda or Folktale, which offer utility functions and **monads**. The resulting code looks quite clean.

Look at the code below. A teacher uses Javascript and Folktale's **Result Monad**, which is often called **Either** in other languages/frameworks, to print a congratulatory message.

You are not expected to understand everything in the example yet. Explaining this stuff is the whole point of this book.

```javascript
// import the result monad from folktale
const Result = require('folktale/result');

// no magic numbers
const MINIMUM_SCORE = 0;
const MAXIMUM_SCORE = 20;

const checkValueBetween = (lower, upper) => (value) => {
    return value >= lower && value <= upper ?
        Result.Ok(value) : Result.Error('Value was negative');
};

const calculatePercentage = (max) => (value) => value / max * 100;

const congratulation = (value) => `Congratulations! You scored ${value}%`;

// setup of methods via currying / partial application
const betweenZeroAndTwenty = checkValueBetween(MINIMUM_SCORE, MAXIMUM_SCORE);

const calculatePercentageMaxPoints20 = calculatePercentage(MAXIMUM_SCORE);

// workflow / pipeline / railway
function printCongratulationWithPointsAsPercentage(value) {
    const result = betweenZeroAndTwenty(value)
        .map(calculatePercentageMaxPoints20)
        .map(congratulation)
        .merge();
    console.log(result);
}

// examples
printCongratulationWithPointsAsPercentage(12); (1)
printCongratulationWithPointsAsPercentage(-1); (2)
```

(1) This prints "Congratulations! You scored 60%".
(2) And this prints "Value was negative".

Just to be clear, the `(1)` and `(2)` should be removed if you want the above to compile. I will use numbers between brackets when I want to comment on something in the code. Also, I prefer removing the braces for short functions like those above. But that's not very readable in a book. So for clarity I sometimes keep the braces and split up code over multiple lines. Similarly, I sometimes pick shorter, less informative names for functions and parameters to save space. But even then, my code is sometimes too long and does not fit on these pages! So if you see a `\` at the end of a line of code, it probably means the line was too long and part of it was placed on the next line.

What's so nice about this? First, the entire logic consists of **small**, **reusable functions**. Right now they are configured for scores with a maximum score of 20, but that can be changed by fiddling with the MAXIMUM_SCORE constant. In a more complete example, we might put that value in the configuration. Two of the first three methods are highly **composable**. Using a helper, we can compose two of the functions and get the result we expected:

```
// helper function for composing
const compose = (...fns) => (...args) => {
    return fns.reduceRight((res, fn) => [fn.call(null, ...res)], args)[0];
};

const composed = compose(congratulation, calculatePercentageMaxPoints20);
composed(10); (1)
```

(1) This returns "Congratulations! You scored 50%"

This is because they are both **pure** functions. Our 'check' or validation function is also pure, though slightly more difficult to use. But because it returns a monad we can use `map` to pass its value to the other functions.

Monads, maps, purity, testing and most of the other terms mentioned here will be discussed in upcoming chapters.

So we have clean, short functions which are easy to test and can be used to build larger components, as if they were pieces of Lego. Another nice thing is the exception handling. Instead of throwing exceptions when a value is invalid, we make note of the issue and continue. And when we are ready, we handle those errors in whatever way we see fit. But we don't break control flow just because something was slightly amiss. In my eyes, this kind of functional code is elegant, concise and understandable. But unfamiliarity might make it a bit harder to read. Give it some time. Some two hundred pages worth of time.

So this could have been the introduction to a book called *Functional Programing In Javascript*, featuring a library like Folktale or Ramda. Instead, it's about Typescript.

Why Typescript?

Here I might be entering flamewar territory. So I will first note that I have learned to like writing code in dynamically-typed languages. There is less typing - *pun intended* - involved and combining functions is easy because the language believes you know what you are doing. And if your test coverage is high enough, you can avoid most of the strange bugs that can occur because of the lack of typing.

Even so, strong typing has several advantages which in my experience are even more outspoken when you are working in a team. Which, again, is important to me because I want to write functional code on the job and that means working in some sort of agile team. What are those advantages? Well, types are more beginner friendly, meaning they help both less experienced colleagues AND developers who are new to your team. The latter might know the language but will be unfamiliar with the domain and code. Types tell your team members more about how the function can be used. Take a look at the following Typescript function.

```
function sayHello(value: string): string {
    // implementation here
}
```

Already we know several things about this function. The return type is string so presumably the function will not say 'hello' by printing to the console. Instead it will return its greeting. We can also guess that it will be a personalised greeting, because we have a string parameter with the generic name `value`. Perhaps it will contain a name, or a title? But probably not an age, because in that case number would be better suited. And yes, the programmer could have chosen a better parameter name. But code is not perfect. So types can still offer additional information.

Another Typescript example:

```
type Person = {
    name: string,
    age: number,
};

function getAge(person: Person) {
    return person.age;
}
```

The Person type helps a new developer a lot. If someone asks him to write a function which returns the name of a person in uppercase, he knows that a person with the appropriate type will have a name field. And since it is a string he can use `toUpperCase()` on it. Without types, he would have to

a. find out what the name field was called

b. hope he does not make a typo and
c. pray that the value of the field will actually be a string.

Yes, tests would help as well. But we are talking about new functionality for which no tests exist yet. So they would not help the new developer find his way around the code base.

Modern IDE's help you write both statically and dynamically typed code. But they always have better support for typed languages. For Java, an IDE like IntelliJ will help you in hundreds of different ways. Give the wrong type of argument to a function or method, and it will warn you and offer ways to fix the mistake. If you do not return a value at the end of a method, you will get a warning and a suggestion to change the method's return type to `void`. Reading code also becomes easier. Just click around. Your IDE knows where it has to go because it sees that the given method is part of interface X which is implemented by Y. When working in a dynamically typed language your IDE might let you decide between a few or a dozen of candidate files. Maybe the piece of functionality you are looking for is among them. Maybe not. Again, this help will be especially useful to new developers. You might know that `toUpperCase()` is a string method. But they might not.

A related advantage of types is that types serve as documentation. In a good project, types can tell you what kind of data will enter your application, the way that data is transformed by your code and what information you give back to the outside world. Tests similarly provide a form of runnable documentation, but they have a different focus. Types center on the business events your code has to deal with, while tests play out scenarios with these events. So having both types and tests is better than only having one of both.

Interestingly, types can be used to "*make illegal states unrepresentable*". A simple example of this is the `age` field in Person, as seen earlier. In our code, `age` is always a number. We do not want to allow string values like '22' in our domain. *And the type has made this illegal value impossible to represent*. It is a simple example, to be sure, and we can still give age a lot of illegal values. -10 and 1239473 may not be numbers we expect to find when Person in our domain represents a human adult. Which means that making illegal states unrepresentable requires both creativity *and* a powerful type system.

Which brings me to a realisation I had last year - which others have had before me. I started out as a Java developer. When I was first confronted professionally with a Javascript/Node project, the qualities of strong typing which I just discussed really came to the forefront in my mind. Once I got used to writing Javascript, the qualities of the dynamic moved to the forefront. That such a shift in thinking came fluently probably explains why I like the "*strong opinions weakly held*" quote. Anyway, by that time I had noticed the downsides of types and thought I did not need them anymore. As a more experienced developer, dynamic languages were more my thing.

After encountering Haskell, I came to a different realisation. What had bothered me about Java was that a) I had to do a lot of stupid typing and b) the types I wrote were often really obvious and c) despite all that work, the types were still letting a lot of stupid mistakes slip by. So it wasn't the types that were bothering me, *it was the lack of a powerful type system with good inference*. If Java had Haskell-like inference I would not have had to write all those bloody types. If it had **sum types** like those of Haskell, I could express more powerful ideas more elegantly and banish more illegal

states! With both a really powerful type system and great inference you get a lot of safety for free! For no additional effort, you will almost always know that your code will actually run if it compiles. Which is awesome.

I keep picking on Java because it is a language I have a lot of experience with. So its qualities and weaknesses come to mind more easily than those of C#, Go, etc. In any case, the good news is Java keeps on adding new useful features, fixing more and more of its shortcomings.

Back to Typescript, which is the most popular way of adding types to Javascript. It is a superset of Javascript, which means it has shortcomings because it needs to take into account how Javascript behaves. Good types are easiest when you are building something from the ground up… In addition, circumventing the types is not very difficult and the strictness of typing depends on configuration settings. But that should not be an issue if your team is committed to making their code as safe as possible.

Typescript takes a very lightweight approach to adding types. They only exist at compile time, meaning they are not present in the output, which is just plain - and pretty readable - Javascript. So there is no runtime overhead. Meanwhile the type system itself is quite capable. For example, when we have a function that lets a bad guy say hello, we could limit the name parameter to the string value of our bad guys names, stopping faulty names from being entered - at compile time. This is one of the powers of type systems that we talked about earlier: we are making illegal values unrepresentable. Only bad guys can be passed to the function. A nice guy like Minsk is simply incapable of saying hello in an evil way.

```
type GoodGuy = 'Minsk';
type BadGuy = 'Irenicus' | 'Bodhi';

function helloInBadWay(badGuy: BadGuy) {
    return `${badGuy} says hello to the child of Bhaal.`
}

helloInBadWay('Irenicus'); (1)
// helloInBadWay('Minsk'); (2)
```

(1) This will work
(2) But this will not compile and give you the following warning: *Argument of type '"Minsk"' is not assignable to parameter of type 'BadGuy'.*

And Typescript allows for more complex types as well:

```
type Fighter = {
    _type: 'fighter',
    health: int,
    weapons: string[],
}
type Wizard = {
    _type: 'wizard',
    health: int,
    spells: string[],
}
type Cleric = {
    _type: 'cleric',
    health: int,
    prayers: string[],
}

type Classes = Fighter | Wizard | Cleric; (1)
```

(1) Your class is either fighter, wizard or cleric in this game. Each has a number indicating health, but their other properties vary.

Meanwhile you can find examples on the internet of people using Typescript 4.1 to verify the correctness of a string that needs to form a path to a possibly nested property of an object! With every new release (there are multiple every year) new abilities like these are added. Personally, I hope regex types[2] will be among those capabilities at some point in the future.

So Typescript has a powerful type system. That's a start. But it also has that other element that I mentioned: type inference. Sure, that inference is limited when compared to that of a language like Haskell. And it is advised to add explicit types to your returns. But it still adds a lot of value. Try it! You might be surprised how quickly you get used to it. The combined value of the type system and its inference goes a long way in explaining the steadily-growing popularity of Typescript.

Why FP-TS?

So Typescript adds types. Do we need anything else? Why, yes. Even though Javascript is one of the easier mainstream languages to write elegant functional code in, it lacks several useful, even essential features. So we will need some additional libraries/frameworks if we want to write 'real' functional code. The kind of stuff we can deploy to production.

I already mentioned Ramda and Folktale. But they are unfortunately insufficient. Ramda lacks the extremely useful concept of monads on which several of the more advanced FP techniques are based. Folktale is a small library with some of the most essential monads. But it has very few other features

[2] https://github.com/Microsoft/TypeScript/issues/6579

and it does not have some of the more powerful features of FP like **Monad Transformers**. And another downside is that the library has not received a lot of attention in the last few years.

So we will be using a Typescript library called fp-ts[3]. It has several advantages: written in Typescript, many stars on Github, actively maintained (at the time of writing there was a new commit just a day earlier) and powerful monads. Plus, it has an ecosystem[4] of libraries that add even more functionality. The author of the library - and a few other people - have also written blog posts exploring parts of the library's functionality.

One downside is that the documentation is OK, but not great. In fact, the Github page has a warning: "*Teaching functional programming is out of scope of this project, so the documentation assumes you already know what FP is.*"

Motivation for this book

Wait. Teaching FP? That's exactly what this book wants to do! Not that there are no other resources out there. You can find tons of video's, blog posts and articles talking about FP concepts. There are also quite a few books about Haskell and a couple of good ones that talk about these concepts in other languages[5].

But in my experience, most articles and video's only talk about the basic concepts of FP, leaving the rest up to you. Books occasionally go a bit further in explaining how to actually use the paradigm. But it is rare to see concrete larger examples, on the scale of basic - but useful - projects[6].

So we are going to be ambitious here. We will write a practical guide that both *explains the most important concepts of FP in understandable terms* and shows *how to actually use them.*

What you should know

I wrote this book with a public of OOP programmers - with some Javascript experience - in the back of my mind. Absolute beginners will probably find this book hard because I skip over several basic concepts and ideas. On the other hand, multiple years of professional experience are definitely *not* required. Nor is knowledge of Functional Programming. We will be using Amazon Web Services (AWS) in a few chapters but you should be able to follow along without any Cloud background.

Overview

The next chapter contains some useful Javascript and Typescript tidbits and some other general guidance. It can be skipped or skimmed by users with extensive Javascript experience.

[3] https://github.com/gcanti/fp-ts

[4] https://gcanti.github.io/fp-ts/ecosystem/

[5] Like *Functional Programming in JavaScript*, by Luis Atencio

[6] One book I really enjoyed, which talks about using FP for modelling your domain and building applications, is *Domain Modeling Made Functional*, by Scott Wlaschin

The chapter after that explains core FP concepts and show how to use the fp-ts library. After that, you will find chapters with example projects loosely based on real-world applications, further digging into concepts from the functional world and how to use them in practice. After that, we will briefly explore more advanced subjects, like the writing of **DSLs** and **Free Monads**. All these chapters will contain (a lot) of example code and some projects will be deployed in the AWS Cloud. Important terms will often be introduced with **bold** markings. Asides offer optional additional information, warnings, tips.

I have tried to make this book 'complete', in the sense that all relevant code is included. This was done for two reasons: one, a reader does not have to switch between the book and additional code on GitHub. Two, I have had issues with code in books where certain 'non-vital' parts were left out for the sake of brevity. Like imports. Because when you're a beginner in a language/framework/… even knowing what exactly to import can be useful. The idea of having all code in the book has not, however, turned out perfectly. So be prepared for some redundant code, some skipping of test scenarios that seem less relevant, the leaving out of imports if they were already shown in earlier pages, etc. All of this to avoid needlessly adding to the length of the book and boring the reader.

Tidbits: Some Useful Javascript and Typescript

This book expects some basic knowledge of Typescript and Javascript from its readers. So we will not be explaining concepts like functions and variables. But for those who have less day-to-day experience with the language, the following tidbits might be useful. The chapter can be skipped or skimmed by more experienced programmers.

Requirements

You should have node and npm installed on your computer, though the latter should be available if you have the former installed. You can check whether both are available by opening a terminal window and typing `node -v` and `npm -v`. If they are installed, you should get back a version (for example `v15.5.0`).

Setting up a new project

If you want to follow along with the examples, add a new directory to your computer and run `npm init` to create a new project. You can skip through most of the questions. For the Javascript examples, nothing special is needed. Once we get to fp-ts, you will need to add both Typescript and fp-ts to your dependencies, in addition to some other dependencies that will be noted later on. Typescript, and types packages (which add types to existing Javascript libraries), can be placed under dev dependencies because they are not needed at runtime. So for now your `package.json` file will look like this:

```
{
  "name": "fpintypescript",
  "version": "1.0.0",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "dependencies": {
    "fp-ts": "^2.8.6"
  },
  "devDependencies": {
    "typescript": "^4.0.5"
```

```
  },
  "license": "ISC"
}
```

To add new dependencies later on, just find the package on npmjs.com and run the install command shown there. Also create an `src` folder inside your project directory. This is where you will put your Typescript (.ts) files.

As we mentioned in the introduction, Typescript files have to be compiled, which you can do by running `tsc` in your terminal (it is also possible that your IDE will automatically compile ts files for you). If you added Typescript to your dependencies, the `tsc` command should be available within your project. To run it anywhere, install Typescript globally: `npm install -g typescript`.

The `tsc --init` command can be used to generate a `tsconfig.json` in the root of your project. This file tells Typescript what Javascript version to compile to, where to output Javascript files, etc. For the first few chapters, a generated tsconfig might be all you need *but be sure to set the 'target' no lower than es6*. Later on we will use webpack to build our output and show you the `tsconfig` that goes with that setup.

More than one way to build a function

You should already know the basic way to create a function, using the `function` keyword. The following, when called, will print a message to the console.

```
function example() {
    console.log('Print a message');
}
```

Now since functions are first-class citizens in Javascript, we can also do things like creating a function that creates functions.

```
function withInnerFunction(outer) {
    return function ourInnerFunction(inner) {
        console.log(`Outer function received ${outer}, inner ${inner}`);
    };
}

const returnedFunction = withInnerFunction('a'); (1)
// some time later in our program, we actually need to call returnedFunction
returnedFunction('b'); (2)
```

(1) A function is returned by calling withInnerFunction and saved in the returnedFunction variable.
(2) When we call the returnedFunction function it prints *"Outer function received a, inner got b"*.

The above is called a **closure**. With closures, the inner function has access to all the information available in the outer function. Which is why the value of parameter `outer` is still available when we decide to call the inner function that was returned when we called `withInnerFunction`. This is interesting because it allows us to pass additional information (which might serve as environment or state) into the function we create. Closures and first-class functions allow for loads more interesting behavior, like creating factories for building functions, keeping parameter lists short, reducing code duplication, etc. Also, closures are about the only way of making a variable really private in Javascript. All the other ways can be circumvented.

There is a downside to the above way of creating functions within functions though. It quickly becomes noisy. And Javascript likes strange and quirky stuff, but not noisy. So there is another way of declaring functions. Look at the following code. The first function is declared in the traditional way. The second uses `const` to declare a function. Notice that we do not have to use the `function` keyword. The last example shows how we can leave behind the brackets for shorter functions (this is called an **implicit return**), leading to a nice one-liner.

```
function example() {
    console.log('Print a message');
}

const newExample = () => {
    console.log('Print a message');
};

const newExampleAlt = () => console.log('Print a message');
```

The second and third declaration are examples of **arrow functions**. They are largely the same as normal functions, except when it comes to the `this` keyword, which we will not be using in this book.

So the last example is slightly shorter and less noisy. It also helps you write cleaner nested functions. Let us use an arrow function to rewrite `withInnerFunction`. I abbreviated the `console.log` message a bit.

```
const withInnerFunction = (outer) => (inner) => console.log('Outer function...');
```

We changed four lines into one, removed the braces and the `function` keyword, which appeared twice. And this is only a toy example with one nested function. In real code, you will occasionally go several levels deeper. Note that there may be some impact on performance when you have extremely deep closures (10, 20 or more levels).

Backticks for easy printing

There are several ways of composing strings and printing them to output. The old way is appending using the plus sign:

```
const w = 'world';
const hw = 'hello ' + w;
console.log(hw); // "hello world"
```

Which becomes a mess in more complex cases:

```
function example(one, two, three, four) {
    return 'We received ' + one + ', also ' + two + ' fourth was ' + four + ', third\
 was ' + three;
}
```

In almost all cases it is briefer and clearer to use backticks, which allow you to specify the variables inside the string.

```
function example(one, two, three, four) {
    return `We received ${one}, also ${two} fourth was ${four}, third was ${three}`;
}
```

The ternary operator

All languages offer some variation on the `if/else` structure. Because we all have choices to make.

```
function choices(thirsty) {
    if(thirsty) {
        return 'beer';
    } else {
        return 'hamburger';
    }
}

console.log(choices(true));
```

The above will print "beer" when thirsty is 'truthy', or "hamburger" if you pass a `false`.

Some languages are very strict about their booleans. Java and C#, for example, would require a boolean for thirsty and would reject anything else. Other languages, like Python and Javascript are - true to their dynamic nature - more flexible and only require the values to be truthy. Which means that they will be converted from the original value into a boolean. What becomes true and what becomes false depends on the type of value. Null and undefined are, perhaps obviously, false. An empty string is false as well, while all other strings are true. For numbers, 0 is false. Most of these conversions make sense and they can make the code cleaner and more concise. On the other hand, truthiness can be dangerous when the value you pass in is not what you thought it was (for example, if you thought you were dealing with a real boolean but in fact are passing in a string).

Most Javascript programmers and linters will shorten it to the equivalent:

```
function choices(thirsty) {
    if(thirsty) {
        return 'beer';
    }
    return 'hamburger';
}
```

`If` statements such as these are fine but they have one downside. They are **statements, not expressions**. An expression *produces a value*. But a statement is not required to produce anything.

```
function choices(thirsty) {
    if(thirsty) {
        console.log('beer');
    } else {
        return 'hamburger';
    }
}
```

This rewrite will return undefined for a true thirsty value, but its 'shortened' version (where we again leave behind the `else`) will print "beer" if the value is true AND will always return the string "hamburger". The first version is especially unfortunate, because we now have one path that returns a string and the other that has a missing value. Javascript linters like Eslint have rules specifically for avoiding this confusing type of behavior.

This is a downside of statements that is not liked in FP. Functional programmers like *all* their paths to return values thank-you-very-much. Luckily, many languages offer an alternative `if/else` that serves as an expression. In Python it is a variation on the normal `if/else`, in Haskell you have an `if-then-else`. And in Java, Javascript and other languages you have a ternary operator:

```
// will always return something
const choices = (thirsty) => thirsty ? 'beer' : 'hamburger';
```

It is nice and short, plus it will complain if you do not have a value to give back for one of the paths. When (deeply) nested it can become unreadable - but maybe that just means your function is doing too much and you should split it up.

Discriminating unions

Algebraic data types (ADTs) in FP - which will be discussed later on - can be things of beauty. In Typescript, we can use **discriminating unions** (also called **tagged unions**) to get **sum types**, which are a category of ADTs. Creating them does require a bit of ceremony though.

From the Typescript Handbook comes the following example:

```typescript
type NetworkLoadingState = {
  state: "loading";
};

type NetworkFailedState = {
  state: "failed";
  code: number;
};

type NetworkSuccessState = {
  state: "success";
  response: {
    title: string;
    duration: number;
    summary: string;
  };
};

type NetworkState = NetworkLoadingState | NetworkFailedState | NetworkSuccessState;
```

From this we know our network can be in one of three states. It is either loading, succeeded or failed. Each state has different properties associated with it. With a failure comes a `code`, which can show the type of error that occurred, while success contains a `response`. The exclusivity of the sum type is nice. Imagine we had some type of flag indicating the State and lumped all fields together. We could by accident create a `NetworkFailedState` with fields that actually belong to a `NetworkSuccessState`! The type would also be a mess to read, with many fields, most of them optional. A sum type makes the State cleaner and easier to read.

In a language like Haskell, you can do very powerful pattern matching on sum types. There are libraries in the fp-ts ecosystem that try to add some of this power to Typescript (for example, morphic-ts). But Typescript can already do some cool things with these unions. When you do checks on the 'tag' field (a field which can be used to differentiate the types within a tagged union - hence the name), it will correctly deduce what specific type you are dealing with. Plus, when using a switch it can do an 'exhaustiveness checks'. The `assertNever` default below will not do anything *if we cover all cases*. Were we to add a fourth kind of state, it would tell us that this new state cannot be assigned to a parameter of type `never`, which is what the default fallback is returning.

```
function assertNever(x: never): never {
  throw new Error("Unexpected object: " + x);
}

function logger(s: NetworkState): string {
  switch (s.state) {
    case "loading":
      return "loading request";
    case "failed":
      return `failed with code ${s.code}`;
    case "success":
      return "got response";
    default:
      return assertNever(s); // fails if we don't handle every possible state
  }
}
```

This is a nice example of how the type system can help us in writing and refactoring code. Also note how Typescript did not fail on the `${s.code}` part. Thanks to the check on state it knows that the type is actually `NetworkFailedState`, which has a `code` field.

Summary

In this chapter, we gave some advice on how to set up a project for following along with the code in this book. We also discussed various ways of writing Javascript code that will be used in this book, but might not be all that familiar to the reader, including arrow functions, backticks and ternary operators. We also briefly discussed discriminating unions from Typescript.

We are now ready to learn more about Functional Programming.

First Steps with Functional Programming

In this chapter we will give an (incomplete) overview of the ideas behind functional programming. We will also introduce some monads as they appear in the fp-ts library and use some of its helper functions.

What is Functional Programming

Functional Programming is a style of programming that places functions front and center. Which is similar to how Object Oriented Programming has classes at its core. Obviously, this means that in FP functions should be first-class citizens. This is a term we already encountered in earlier chapters. Simply put, it means that functions can be used and passed around as data. You can bind functions to variables, you can pass them as arguments to other functions, you can return them from a function, etc. Also note that functions receiving or returning other functions are called **higher-order** functions.

Pure functions, referential transparency and immutability

FP wants its functions to have a couple of qualities. All - or as many as possible - functions should be **pure**. A pure function is one whose result is based solely on the input it received. So as long as the input does not change, the output does not change either. This means it has no **side-effects**, which are any effects that your code does 'on the side', besides its main task of taking in parameters and returning results. This large category includes calls to a database, I/O, but also logging. Why no side effects? Because with them your function is no longer guaranteed to return a certain output for a given input. At some point, that call to your database is going to fail and your code will throw an error instead of returning normally.

Pure functions have several nice qualities. They are easy to (unit) test because they only depend on what you pass in as parameters. They are also easy to reason about, mainly because the lack of side effects makes them simpler and more predictable. A related concept, **referential transparency**, also helps us reason about code. It basically means we can replace a function call with the value it produces. For example, say we have a function that increments a given number by one.

```
function addOne(el) {
    return el + 1;
}

// this
function biggerProgram() {
    const result = addOne(5);
    return result * 2;
}

// is equivalent to this
function biggerProgramAlt() {
    const result = 6;
    return result * 2;
}
```

Several of the examples in this chapter will be written in Javascript, not Typescript. This for both brevity's sake and because in these simple cases types add nothing worthwhile to the example. Starting next chapter, it will be Typescript all the way I promise!

We replaced the call to `addOne` with its result (5 + 1, so 6) and we get the same result. The lack of side effects and referential transparency means that pure functions **compose** easily, an important quality. But more on that later.

Immutability is another often mentioned element of FP. Instead of changing our existing data structures, we create new data structures whenever we change any information. This makes reasoning easier. There is no way that the array you just received from your function was changed by some other process because it is an entirely new data structure! Another obvious advantage of immutability is when you are faced with parallel programming because, paraphrasing Adam Smith, threads coming together with mutable data ends in a conspiracy against your sanity.

I recently helped a colleague fix a bug with some tests. Whenever he ran them individually, everything was fine. But when they ran together some tests mysteriously failed. Our first suspects were improperly destroyed mocks, but those all checked out. After some investigating, I discovered that the parameters that the functions were receiving in the test setup were `const` variables that contained arrays. And the 'system under test' was using `shift()` to get the first element of those arrays. And `shift` mutates an array in place… Combine that with multiple tests using these parameters and we had ourselves a problem. So even when you are not doing any fancy parallel programming, stupid but easily missed mutability bugs can waste your time.

In some cases, especially when your language was not written for immutability, performance may suffer. As always this is a matter of trade-offs. Is excellent performance essential, or is the

combination of a lack of bugs, easier maintenance and decent performance acceptable? And yes, Javascript was sadly not written with immutability in mind. Hence why there are several libraries offering to add immutable data structures, which is obviously inferior to having the language take care of everything for you.

Writing functional code

A question that has gone unanswered until now is how we combine our small, focused functions to build larger applications. In OOP, you create applications by letting objects, containing state and behavior, interact with each other. What does FP offer instead?

First off, a more declarative style of programming. OOP can have an imperative feel to them. Your program consists of a list of instructions to execute (or for your objects to execute). FP tries to lift this sort of behavior to a higher level of abstraction. Whenever possible, the low-level behavior is left to the computer. A well-known example of this is how to execute repeated instructions. In imperative programming, this is often done using loops.

```
const exampleArray = [1, 2, 3, 4];

function addOneToEach(arr) {
    for(let i = 0; i < arr.length; i++) {
        arr[i] = arr[i] + 1;
    }
    return arr;
}

addOneToEach(exampleArray); // [2, 3, 4, 5]
```

But this is repetitive, error-prone and adds a lot of boilerplate. So FP prefers **recursion** or higher-order functions like `map`, which is actually built-in within Javascript for array and executes a function for every element of that array.

```
const exampleArray = [1, 2, 3, 4];

function fpAddOneToEach(arr) {
    return arr.map(el => el + 1); // add 1 to each element
}

fpAddOneToEach(exampleArray); // [2, 3, 4, 5] again
```

An additional advantage of higher-order functions is that they often use immutable data structures. The loop in the first example, on the other hand, mutates the existing array. Which is error-prone,

with me even running into issues when writing the above example code. (I was calling the loop version followed by the FP version and was getting [3, 4, 5, 6] as my unexpected result, because the example array had been modified.)

Also, about that composition. Remember our `addOne` from an earlier example? Well, using it within `map` is trivial:

```
function fpAddOneToEach(arr) {
    return arr.map(addOne);
}
```

The above is called **point-free style** or **tacit programming**. The function we passed in expects one argument and `map` produces one. So we don't have to write `el => addOne(el)` because Javascript knows it has to pass the incoming value to the function.

So with higher-order functions, like filters and maps, we only have to write logic that filters/transforms a single element. The rest is taken care of by the higher-order function. We know how to increment a single value and without writing any additional code, we can now handle arrays!

Additionally, in functional languages maps are available for more than just arrays, thus further extending the usefulness of this function. And because functions like map and filter are more abstract and less bound to specifics, they can be applied in many situations. They will work with any kind of array you throw at them. Whereas - somewhat simplified - the methods of objects only know how to do things for one specific class. Writing functions that operate on a broad range of data structures are somewhat of a feature of functional language like Clojure and Haskell.

The result is also clearer/more readable because filter, map, reduce, fold… all have a clear purpose. You can do anything in a loop. Filter has one job - meaning you already know what it will be doing just by glancing over the code. Try doing that with a non-trivial loop! Compare the following two examples. They do the same thing, but the first runs in a loop and the second uses filters and map.

```
// filter out any elements smaller than 2. increment what remains
// add them to a new array if the result is larger than 4
// return the new array
function inALoop(arr) {
    let newArr = [];

    for(let i = 0; i < arr.length; i++) {
        if(arr[i] > 2) {
            let temporaryResult = arr[i] + 1;

            if(temporaryResult > 4) {
                newArr.push(temporaryResult);
```

```
            }
        }
    }
    return newArr;
}

// do you really need an explanatory comment for this one?
function withFilterAndMap(arr) {
    return arr
        .filter(el => el > 2)
        .map(el => el + 1)
        .filter(result => result > 4);
}
```

The second example is shorter and, if you know what filter and map do, a lot clearer. It can be *read* whereas the loop has to be *deciphered*. A programmer experienced in the first technique can become a very capable decipherer, but it will always take more effort than just reading plain English.

And this does not seem to be a fringe opinion, because many languages have added functions like the above as an alternative to imperative loops. And Java became one of those languages in version 8, with the Streams Api. The corresponding Javadoc, notes the following[7]:

Stream pipeline results may be nondeterministic or incorrect if the behavioral parameters to the stream operations are stateful.

And (let's limit ourselves to only two examples):

Side-effects in behavioral parameters to stream operations are, in general, discouraged, as they can often lead to unwitting violations of the statelessness requirement, as well as other thread-safety hazards.

So what they are saying is: *choose pure functions for reliable behavior* when using Streams. They note in several locations that this is also a requirement for parallelism, for reasons that are no longer a mystery to us: pure functions behave in a predictable fashion. But with side-effects and mutations, anything goes.

The other way FP deals with repetition and avoiding loops, and which should be familiar to anyone reading this book, is recursion. It is solving a problem by breaking it up in smaller versions of the problem in which the solution is simpler. For example, a recursive function for calculating the sum of an array is:

[7] https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/stream/package-summary.html

```
function sum(arr) {
    if (arr.length === 0) return 0; (1)
    const [head, ...tail] = arr; (2)
    return head + sum(tail); (3)
}
```

(1) This is our 'terminal condition'. If we have an empty array, there is nothing left to count, so we return 0.
(2) Here we take the head of the array, its first element, and separate it from the remaining elements (zero or more) which we put in a variable called `tail`.
(3) Now we add the head to the `sum` of the remaining elements. Our function sum will be called repeatedly until we reach terminal condition (1).

Say we call the above with an array with 3 elements: [10, 20, 30]. The function will take the first element, the number 10 and add it to the sum of the tail, i.e. the sum of array [20, 30]. What is that sum? Well, we go into the function again and find out it is the sum of 20 plus the sum of array [30]. And that function is the sum of 30 plus and the sum of an empty array, which - as we know from (1) - is 0. So that returns 30 + 0 ⇒ 30, which is added to 20, giving 50, which is added to 10, giving 60, the correct end result.

Recursion often leads to elegant solutions for complicated problems. But it is not frequently used in mainstream languages like Javascript because the repeated function calls cause the stack to grow, possibly resulting in a stack overflow. (Though there are ways around this, through use of a trampoline for example.) In this book we will mainly use higher-order functions to solve problems involving repetition.

Writing functional programs

Getting back to the topic of building applications: higher-order functions are the first way of building larger programs out of smaller components. In `withFilterAndMap` (above) we could have created three one-liner functions, to be used by this function in combinations with the map and filters. Together these produce the same results as a loop of 14 lines. So we get shorter, cleaner and more reusable code (several simple functions compared to a long and very specific loop).

There are other ways of composing functions. First, one that we already touched on briefly, we can combine functions by putting one after the other, with the output of the one becoming the input of the next:

```
const compose = (...fns) => (...args) => {
    return fns.reduceRight((res, fn) => [fn.call(null, ...res)], args)[0];
}; (1)

const coolName = (name) => `${name} a.k.a. Badass`; (2)
const yelling = (name) => name.toUpperCase(); (2)

const yellingACoolerName = compose(yelling, coolName); (3)

const result = yellingACoolerName('sam'); (4)

console.log(result); (5)
```

(1) This helper function already appeared in the introduction. It is simple but ingenious. It receives a rest parameter (zero or more parameters - in this case all functions) and returns a function that takes another rest parameter. The returned function calls `reduceRight` on the first rest parameter. This means we run a reduce and start with the rightmost argument. So you first call the rightmost function that was passed in. We pass in the current result as an argument to the function that is currently in the reduce. And the initial 'result' - *before* we start the real reduce - consists of the args we received as a parameter. Finally, because our reduce returns our result within an array, we pick the first (and only) element.
(2) A few simple functions, one for making a name cooler and one for yelling it.
(3) Now we pass these two functions to `compose`. This returns a function waiting for additional arguments.
(4) Now the process discussed in (1) kicks into action. We start with a reduceRight on our array of functions. First up is `coolName`, which receives the parameter 'sam' as the preliminary result. The function is called and adds a cool suffix to the name. Next, our reduce starts working on the yelling function. It too receives a preliminary result, which is the name *with the suffix that was just added*. It returns this parameter uppercased.
(5) The result is printed to the console: "SAM A.K.A. BADASS".

Cool, right? No, not the name we just printed - the way we created new functionality out of existing functions, similar to what we did with map and filter. And composition has several nice qualities to it. For one, we did not have to change our 'existing' functions. Instead our new functionality was created out of existing code. And if that code was already in use and tested, we know the individual pieces work. With only a little practice we can start reasoning about the composed functionality: "ok, so first the argument is made cool, next it becomes uppercase… yep, looks good". And once we add Typescript, types will warn us if we try to compose functions that do not go together! Also, if the original functions are pure, the result is pure as well. So everything will be easy to test. As a small side note, the order of compose (right to left) is probably inspired by mathematics: if you have a function `f(g(x))` you can first calculate `g(x)` and pass result y to the outer function: `f(y)`.

The one thing we might argue about is whether the larger functionality is easier or harder to understand than imperative code. In most cases, this will depend on the quality of the functions.

Are there names clear and self-explanatory? Ideally, you want to know what a function does just by looking at its name, optionally aided by a type. Remember, you don't really want to know how the underlying function does what it does. You should just trust it to work (hiding implementation details). However, if you have to dive into all the functions in the compose *every time* you explore the code, I can understand if you find this style hard to read. Though the problem would probably be the same in imperative code. Solutions include: a) better names, b) better abstractions, c) more experience with the code base.

For those that have experience with Linux, this type of composition is similar to the **Unix Philosophy** of piping. Have small programs that do one thing well and use pipes to combine them together to form larger functionality. As those familiar with bash will know, the versatility and power of piping, using only basic unix components, is impressive.

We can also use composition for information hiding. For example, we could decide to only expose `yellingACoolerName`, hiding the functions it is made of. Because we do not use classes, modules are our most important means when it comes to hiding information. Modules should only expose the functions that other parts of your program need. A single file which exposes hundreds of functions or modules with ill-defined boundaries will make your code harder to understand.

Let us look at one more example, slightly more realistic - and more complicated than before.

```
const decrypt = (m) => `{ "userType": "admin", "message": "${m}" }`; (1)

const userTypeLens = (fallback) => (userInfo) => userInfo.userType || fallback; (2)

const auth = (type) => type === 'admin' ? ({ allow: true }) : ({ allow: false }); (3)

const userTypeLensDefaultNone = userTypeLens('none'); (4)

const authAnswer = compose(
    JSON.stringify,
    auth,
    userTypeLensDefaultNone,
    JSON.parse,
    decrypt,
); (5)

console.log(authAnswer('a message')); (6)
```

(1) This is a function for decrypting incoming messages. It's a very liberal function because it thinks everyone is an admin.
(2) **Lenses** in FP are used to retrieve or change data in a data structure. They make it really easy to modify a single value within immutable, complicated structures. Real lenses are more complicated

than this example though (look for the helper library monocle-ts if you want to use them in your project). We just retrieve a field from the incoming argument and return a fallback if the field is missing.
(3) This function checks the user type and returns an `{ allow: true }` when the user is an admin. For everyone else, it's false.
(4) We fill in our fallback/default value, which is the string `none`.
(5) We bring our functions together. Remember, read from right to left / bottom to top. We decrypt, parse the string into JSON, retrieve the user type, determine whether that user is authorized and stringify the result.
(6) This will print "allow: true" when it's invoked.

Did you notice what we did in (2) and (4)? We wrote a function that, after accepting an argument, returned a new function. And it was this new function that was passed to our mini-program. We can write way more complex examples than the above:

```
1 const curryIt = (first) => (second) => (third) => ... => (final) => {
2     return 'Now returning your values...';
3 }
4 const result = curryIt(1)(2)(3)...('final');
```

Which means we can write a function that, whenever we feed it an argument, returns another function that also accepts an argument and returns a function… until the final argument is passed. When that happens, the code block will be executed. This is called **currying** and it is very popular in functional languages like Haskell and F#. Closely linked to currying is **partial application**, which is feeding a function *part* of the arguments it needs at given and calling the function with the remaining arguments at a later time. Upon which it will execute. Python has a function called `partial` which can help you do exactly that.

Why is this useful? Well, it allows you to add config/dependencies to a function at one point and call it at a later time. You make sure your function has everything it needs to run correctly. From that moment on it is ready, waiting for the final data input which will run it. Also, it can make your code easier to understand when dealing with functions that take a lot of arguments. In the context of composing, there is another reason for currying. Note that in our above example, every function we use in our 'program' (5) requires one argument. Conveniently, all our functions are returning one value (a string, boolean, object,…), which makes them easy to compose.

So composing is easy when you have the following:

Function ONE: accepts a parameter of type A and returns a value type B
Function TWO: accepts a parameter of type B and returns a value type C
Function THREE: accepts a parameter of type C and returns a value type D
*PLUS a value of type A (to start the whole thing)

Note how function TWO needs a value that can be provided by ONE, while THREE needs a value produced by TWO. Which means they can be composed: we run function ONE with the value of type A, which returns a value of type B. Next up is function TWO, which receives the value of type

B and returns a value of type C, which is passed to THREE. This gives us our end result, a value of type D.

In a Haskell/F# like notation, the signatures of the functions would be like:

ONE :: A -> B
TWO :: B -> C
THREE :: C -> D
COMBINED :: A -> D

So if the output of one function matches the input of the other, we can combine them. But what if function TWO needed multiple arguments?

```
function TWO(first, second) {
    if(first && second) {
        return 'ok';
    }
    throw Error('Did not receive two params!');
}
```

Now composing won't work! We are only passing in one parameter to TWO, because that's all we are getting from ONE. So do we give up on composing? No, because we now know currying:

```
const TWO = (first) => (second) => {
    // same as above
}
const twoWhichWeWillPassToThree = TWO('first argument already given');
```

Rewriting the above, we have:

ONE :: A -> C
TWO :: B -> C -> D
THREE :: D -> E
COMBINED :: A -> E

Which could not be combined, so we gave TWO a B, which returned a new function for us to use:

FUNCTION_RETURNED_BY_TWO :: C -> D

Which fits perfectly. So currying can be a huge help when we want to compose functions *whose input and output values are mismatched*. We give what we can, the rest will be filled in later. As a consequence, the final argument of a function is often the data you are operating on, while earlier arguments are things like configuration and dependencies - because most of the time we know what those will be beforehand. But we only know what the data looks like *when we are receiving it*.

Note that types, though they occasionally get in the way, are generally very useful when composing functions. If a function expects type A and you are feeding it a *function that returns A* instead, it will complain. Meanwhile Javascript will accept everything you throw at it and crash and burn at runtime. In our example, try using `userTypeLens` (2) instead of the partially applied variant (4). Javascript will now return an "allow: false", despite receiving an admin.

So now we know how to compose functions to create our software. And we know how currying can help with functions that do not naturally fit together because they have the wrong types/number of parameters. But with Typescript/Javascript we are still going to run into some issues. The most important one might be that a lot of stuff happens asynchronously. Take a look at the following:

```
1 const givesAPromise = (num) => Promise.resolve(num);
2
3 const doubleIt = (num) => num * 2;
4
5 // uses the compose we defined earlier
6 const doubled = compose(doubleIt, givesAPromise);
7
8 console.log(doubled(2)); // prints NaN
```

We tried to use our trusted `compose`, but one of our functions is returning a Promise instead of the concrete result we want to work with. How do we fix it? If you want to write pure Javascript you could use `then()`, which is provided by Promise, to retrieve a result to work with. Which is certainly not a bad idea. But you might run into some limits and, more generally, your entire program now has to use Promises everywhere.

There are other issues we might run into when composing. What should we do when a function does not return the value we expected? Exceptions might be thrown instead of a function returning cleanly! For this reason, exceptions are not well-liked within FP. Functions should return values so other functions can use those outputs as input. Exceptions meanwhile break control flow. They might cause your program to come to a sudden halt or they can end up caught in an entirely different part of your program. Meaning they make your logic harder to reason about. They have their uses, especially in a panic situation, i.e. when your program has somehow wound up in an unmanageable, corrupt state. In that case stopping everything is often the most sensible course of action.

Go has a keyword `panic` for exactly this use case. Preferably, it is only thrown when you *really* have to stop the program dead in its tracks, often because you're not sure how you would ever recover from the given issue. Errors, on the other hand, are normal values that will be `nil` if everything went according to plan. If they are not empty, you will have to take action, like returning a message to the user informing him of the failure.

But using them for a thousand different common and very reasonable situations, like handling invalid or incomplete user data, does not make much sense. Why would you create a different path

for such cases instead of just handling the situation inside your current function/module? Again, just to be clear, the use of exceptions in your code might be perfectly fine but they are a less than ideal instrument for handling several situations in which they are popularly used. Similarly, several decades ago, you could build a great application using GOTO statements. But most programmers today avoid GOTO because there are better, cleaner ways of structuring your code.

But how can we avoid exceptions? How will we handle error cases without them? And if Promises are not ideal for asynchronous behavior, what should we use instead? Enter the dreaded **monads**!

Monads

There are many articles explaining monads, some from their mathematical beginnings, others using a broad range of examples. Here it might suffice to compare them to a container. We have a certain value (a string, a number, an object) and we put it inside that container. In several functional languages, putting the value inside the monad is called `return`, but since this word is a keyword in Javascript, you will find various other words for it like `of`.

The various 'constructors' for monads in fp-ts are called **lifting functions** for reasons that will become clear in the next chapter. Also, the comparison to a container is disliked by some in the FP world, probably because it oversimplifies things. On the other hand, it is easy to grasp and useful in most cases.

Ok, so we have a value inside a container. What does that get us? Well, monads have several functions that influence/change that value. For example, the monad function `map`, which takes a function that might be applied to the value inside the container. Why did I say *might*? Well, because it depends.

Technically, monads only define a `bind/chain/flatmap`, not a `map`, which is part of the definition of **Functor**. But monads are functors, so our current explanation is good enough.

An example might make things easier to understand. Let us take a look at the Monad **Option** from fp-ts.

```
import {getOrElse, map, none, some} from "fp-ts/lib/Option"; (1)

const upperCaseIt = (value: string) => value.toUpperCase(); (2)

const optionWithAString = some('a value'); (3)
const optionEmpty = none; (4)

const upperCased = map(upperCaseIt)(optionWithAString); (5)
const upperCasedEmpty = map(upperCaseIt)(optionEmpty); (6)
```

```

console.log(getOrElse(() => 'no value present')(upperCased)); (7)
console.log(getOrElse(() => 'no value present')(upperCasedEmpty)); (8)
```

(1) We import the functions we need from fp-ts.
(2) This function turns the string it receives into an uppercase version. Nothing special.
(3) This is more interesting. We place a string value inside the Option Monad using the `some` function (which is a sort of constructor, an `of`).
(4) We can also tell Option that we do not currently have any values to give to it (hence the name Option). It *can* have a value but it might not. So we tell it we have nothing by passing `none`, which is a constant representing an empty Optional.
(5) Now we call `map` and pass it the function we defined in (2) as a first argument. The second argument is the value we created in (3).
(6) Here we pass in the same function plus the empty value from (4).
(7) Now we want to log the value contained inside the monad. Since it is wrapped inside our container, we need a helper to get it out. `getOrElse` will retrieve the value OR use the function you pass in as the first argument as a fallback if the Option turns out to be empty. In this case, our fallback will return the string 'no value present'. Here the result is 'A VALUE', which is printed to output. The value we originally passed in (3) has been uppercased by the function (1) we gave to `map`.
(8) This, however, will print the fallback 'no value present' because our monad was empty.

Ok, so monads are a kind of container for values and Option is a monad that either contains an actual value or is empty inside. We also said that with `map` a function of our choosing can be applied to the value inside our Option. Which happened in (5), as we saw from the printout of (7): the value was indeed uppercased. Equally interesting is that the function was *not applied to our empty container* which was defined at (4) and mapped at (6). Why? Because there was nothing to uppercase!

If we were working with vanilla Javascript, the scenario would be totally different. We would have had a missing value (null) and passed it to a function that uppercases strings:

```
function dangerous(value) {
    return value.toUpperCase();
}

console.log(dangerous(null));
```

And this would have crashed our program at runtime, throwing an ugly exception, *"TypeError: Cannot read property 'toUpperCase' of null"*.

But Option gives us a way out. It offers us the ability to apply functions to values without worrying about how null is going to make it all blow up in our face. If the value is missing (`null` or `undefined`), our code will just sit there, do nothing and give back the same `none` value.

Monads define another function which you may know as `bind`, `chain` or `flatMap`. We will be using `chain` since that is the name used by fp-ts. We will also be ignoring the subtle difference with `flatMap`.

In any case, `chain` is similar to `map` with one important difference. After applying the given function to the value in our container it will flatten the containers. (Hence the name `flatMap`.)

```
import {chain, getOrElse, none, some} from "fp-ts/lib/Option";

const upperCaseSam = (name) => name === 'Sam' ? some(name.toUpperCase()) : none; (1)

const optionWithNameSam = some('Sam'); (2)
const optionWithNameSmith = some('Smith'); (3)

const upperCasedName = chain(upperCaseSam)(optionWithNameSam); (4)
const noneUpperCasedName = chain(upperCaseSam)(optionWithNameSmith); (5)

console.log(getOrElse(() => 'no value present')(upperCasedName)); (6)
console.log(getOrElse(() => 'no value present')(noneUpperCasedName)); (7)
```

(1) This method only uppercases the incoming string if it is equal to 'Sam'. It wraps the result in an Option `some`. It does not like other values, so for those it returns `none` instead.
(2) We created an Option with the string 'Sam'.
(3) And another with the string 'Smith'.
(4) We apply function (1) to the Option at (2).
(5) Similarly (1) is applied to the Option at (3).
(6) We check the result, which is now 'SAM'.
(7) Here, we get a 'no value present', which is the default. So our Option must have been `none`.

We had a container with the value 'Sam' and we applied a function to that inner value which returned *another* container with 'SAM' inside it. I.e. `Option<Option<string>>`. At least, that is what we would have gotten if we had used map. Instead, we used `chain` which flattened out the containers after receiving the results of the function. So `Option<Option<string>>` became `Option<string>`.

Why is this flatten so useful? While monads offer a lot of useful behavior, nesting offers no additional advantages, just more repetitive code since we would have to unwrap the value multiple times. So best to just get rid of the additional wrappings.

In the second case, we entered the function with the value 'Smith' and because this is not equal to 'Sam' our function (1) returned `none`. So our `Option<string>` became an Option with an Option of type `none`. So we have an outer Option, which definitely contains something, i.e. the inner Option. *But that inner Option is itself empty.* And our chain decides that this, once flattened, is equal to `none`. The empty Option 'infects' all of the outer ones. It signifies a missing value, no matter how many times you wrap it with Option.

Let us return to the question of why. Why and when would we use monads? In my understanding, the reason monads were originally created was to handle effects. As we said before, FP likes purity.

But effects like writing to a database or doing a call to some API ruin this purity, because the call might work one moment and fail the next, making everything unpredictable. Enter monads. Because they act as containers, they can 'store' effectful behavior. There are in fact monads for just that, i.e. storing effects, typically called IO. And with `map` we can create functions that further transform the value saved in our container. Now here comes the important thing - at least from an FP perspective: the IO Monad *will not actually create an effect until it is called*. Which means everything is pure *until* you actually call the program containing the IO Monad(s). So we can have our cake and eat it too: effects and purity combined!

While I can appreciate how nice this is, I do think this is a less important aspect of monads for a 'pragmatic' programmer. Luckily, making your programs pure is only one aspect of the advantages monads offer. For example, they make our type system more powerful because they offer more information about what a function is actually doing. A method written in a language like Java or C# will tell the compiler that it will be returning a String, *even if there is a path in which it throws an exception*. In Typescript, you can do this as well.

```
function numAsString(num: number): string {
    if(num === 666) {
        throw new Error('The number of the beast!');
    }
    return num.toString();
}
```

Someone who only sees the type signature of the above might assume that the function will always return the given number as a string. But someday it might throw an error. And nothing prepared you for that. (For me, this is an occasional worry. What if some library I am currently using throws undocumented exceptions?)

Monads help us by being more explicit about what a function will actually do and return. If it has an effect, it is wrapped in an IO Monad. If it is async, it will be an Async/Task Monad (both these names and others are used). If there is a failure path - Either. If the value might not be there - Option. And if it does two or more of these things, we have Monad Transformers for that. In short, more behavior is made explicit and this information helps create better, safer programs.

Monad Transformers are basically monads stacked on top of each other, combining all the useful qualities of the underlying elements. Some Transformers, as well as the monads we just mentioned, will be discussed in subsequent chapters.

In FP, functions for which the input and output of a function are limited in a way that makes sure the signature is (always) correct, are called **total functions**. As an example that fits in nicely with what we just discussed, consider a function that divides the number 10 by a given parameter. What happens when we pass in zero? In some languages this will throw an exception. In Javascript you get undefined and in Typescript you get the special value Infinity. Typescript's answer is perhaps

the most acceptable one, but it might still not be what a user expects from a function returning a number. So we could improve the function by using Option:

```
function divideTenBy(num: number): Option<number> {
    return num !== 0 ? some(10 / num) : none;
}
```

The signature now tells us that we might not return an acceptable answer. If you are in fact a sneaky hacker type and decide to feed it a zero, it will return a `none`, signifying an absence of value. The return is now honest and the function total (at least as far as I can tell, perhaps excluding issues like numbers that are too big for Javascript to handle).

An alternative would be to limit the input type to non-zero values, because for those values we know we can give back sensible results. A helper library for fp-ts, newtype-ts, offers a `NonZeroInteger` type that does just that. All of this is also akin to 'design by contract', a well-known part of the Eiffel programming language, where the description of interfaces is described as a 'contract' which the interface has to obey, with preconditions, postconditions, side effects, etc.

Another thing monads help with is composition. Because every Monad has map and chain functions, we can use them to transform values in a flow/pipeline similar to the one we saw earlier with our `compose` helper. This is also possible with Promises and `then`. But with monads, everything will compose better. Plus we can use different types of monads together to create more powerful functionality. In addition, the difference between chain and map makes what is happening underneath more explicit. I.e. if we see a chain for the asynchronous Task, clearly another asynchronous process is occurring in the function that we passed in.

Another issue with Promises: they do not obey the Left Identity Law. Google that if you are interested in learning more.

Briefly about types again

Throughout the preceding pages, we have seen that types play a role in making our code more safe by stopping certain types of bugs at compile time. Similarly, they can speed up development with faster feedback. Instead of waiting for unit or integration tests, your IDE will immediately tell you that what you are doing is wrong. As another bonus, statically typed languages *can* offer better performance, especially for long-running applications.

Here, I want to get back briefly to the theme of how types help make code more understandable. We already read how being more explicit about input and return types makes code easier to reason about. Instead of - albeit reasonable - guesswork, you know what kind of behavior you can expect. It has been the experience of me and some other developers I have worked with, that in functional programming (or at least code that was more 'functionally oriented') types also help people that did not have the luxury of actually writing the code to understand it and change it. Consider the following:

```
const createPerson = (name) => (age) => (gender) => {
    return {
        name,
        age,
        gender,
    }
} (1)

function printPeople(people) {
    people.forEach(p => console.log(p));
} (2)

const personA = createPerson('firstExampleName')(20)('M'); (3)
const personB = createPerson('thisIsAnother')(24);

printPeople([personA, personB]); (4)
```

(1) A curried function which returns people objects.
(2) A function for printing an array of people to the console.
(3) Create two people.
(4) And pass them to the print function.

If you glance over the code, you might think printPeople will print two objects containing a name, age, gender. And certainly, Javascript has no issues with this code. Everything's good to go! Unfortunately, the second element in the array prints "*Function (anonymous)*" because we failed to pass in a gender parameter! If you did notice the mistake, remember that this is an easy and short piece of code, with everything gathered together in one file. This will not be the case in a real project.

I should note that IntelliJ really does do its best to help. If you hover over `personA` you will see that it is a person whereas `personB` is not. Also, when providing parameters to the createPerson function, it tells you the name of the parameter you have to provide. But even for IntelliJ it is way easier to analyse a single file than all kinds of functions coming from other (internal and external) modules. And in the end, it still accepts that you want to pass a person & a function to the printer - maybe that is really what you want. Compare this to the Typescript version:

```
type Person = {
    name: string,
    age: number,
    gender: string,
}

const createPerson = (name) => (age) => (gender): Person => {
    return {
        name,
```

```
        age,
        gender,
    }
}

function printPeople(people: Person[]) {
    people.forEach(p => console.log(p));
}

const personA = createPerson('firstExampleName')(20)('M');
const personB = createPerson('nameB')(24);

printPeople([personA, personB]);
```

I made sure to add a return type Person to createPerson and told printPeople to only accept arrays of Person. Now I get a warning for Person B: "*Type '(gender: any) ⇒ Person' is missing the following properties from type 'Person': age, gender*". Now our compiler and IDE stop us from going any further because we told it beforehand we wanted to print people, but we are passing in a function instead!

Again, this is a simple example and Typescript is capable of a lot more. And yes, the more complicated the type, the more complicated the accompanying error will be, making things temporarily harder for newer team members. But it is still better than them swearing at tests that won't do as they are told because you passed in the wrong information.

Summary

In this chapter, we looked at some of the most important concepts of FP. We learned about pure functions, higher-order functions, immutability, the role of composition in building applications,… and we took a first look at monads. Most of the examples were brief though, only there to serve as illustrations of the concepts. In the coming chapters, we will start using example projects to explore these concepts and ideas further.

Our First Project: User Registration

In this chapter, we will be expanding our examples from independent programs no more than a page long, to an application that comes a bit closer to the real thing. At this point, the focus is still on getting more familiar with everything we saw in the previous chapter, so we will be ignoring several practical considerations like creating a frontend, deploying our application and saving information in a database. The latter two subjects, and others, will be discussed in later chapters.

The focus on the domain instead of things like persistence also fits in nicely with **Domain Driven Design** (DDD) thinking, which tries to keep things like persistence out of the domain,'at the edge' of our application. That way the domain can remain pure and simple, while we keep our options open and can even switch between databases. Since the core of our application knows nothing about persistence it cares little about whether its data is stored in SQL or noSQL, Postgresql or Oracle! Note that in this book, with its focus on FP concepts, there is no room for any realistic development of a domain and its corresponding **ubiquitous language**.

Defining our types

Our company has decided to add personalisation to its products, so it will now need users to register. We have been tasked with quickly rolling out a first version of this user registration.

Let us try working on this project in the proper way, i.e. postponing our coding for a little bit and first thinking about our requirements in the shape of our types. What does the information we receive look like? What does our user type look like? What functions will we need?

Create a new directory and a `package.json` file with fp-ts included as a dependency (see an earlier chapter for more information about a basic setup). Generate a `tsconfig.json` file with `target` set to es6, or use this one:

tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es6",
    "outDir": "build"
  },
  "exclude": [
    "node_modules"
  ],
```

```
  "include": [
    "src/**/*"
  ]
}
```

Now create a file in the `src` folder called `domain.ts` which will contain all our types. We assume that we will receive a first name, last name, sex/gender, country and age from the frontend/another application. We will call this incoming information `UserRegistrationDto`:

domain.ts

```
export type UserRegistrationDto = {
    firstName: string,
    lastName: string,
    age: number,
    sex: string,
    country: string,
};
```

As someone who often uses AWS Lambda, I actually like to use the suffix 'Event' for incoming data, because Lambdas are run in response to events. In that case, the above would be `UserRegistrationEvent`. DTO, which stands for Data Transfer Object, is another nice suffix, signifying that this is untrusted, outside information. We are still on the edge of our domain/application and transformations need to happen before this DTO becomes a trusted object within our domain.

The core of our little user registration domain is a user. What will that type look like? We could just create something containing all the fields of `UserRegistrationDto`, name it `User` and call it a day. But this is not ideal. Why would we stick with the very rough information and types we receive, when we can improve our types, thus making our lives easier?

So we should try to change the DTO properties into something more limited - and therefore easier for the compiler to work with. This in itself is common practice in DDD. Information from the outside is changed into something suitable for use within the domain. Similarly, anything returned to the outside world is again transformed from 'useful to us' to 'useful to outside parties'. Take a look at the following. We need more code than we would in the naive approach, but the types are more powerful.

domain.ts

```
// user registration dto from before

export type Gender = 'M' | 'F' | 'X'; (1)

export type Europe = {
    readonly _type: 'Europe',
}
export type NorthAmerica = {
    readonly _type: 'NorthAmerica',
}
export type Other = {
    readonly _type: 'Other',
}
export type Region = Europe | NorthAmerica | Other; (2)

export type User = {
    firstName: string,
    lastName: string,
    age: number,
    gender: Gender,
    region: Region,
```

(1) Instead of using sex, we want to use the term gender because we believe it better suits our domain. We may not have the power to change the names of DTO fields (those names might be controlled by another team, might be legacy, etc.) but inside we are free to pick more suitable terms. We assume that gender has to be filled in for a registration to be valid and we limit it to three possible values.
(2) Apparently, our business really focuses on Europe and North America, but exactly what country the user is from does not interest us. So we save a `Region`. If management thinks we can use country in the future, we could add a country field to the region types. But right now this is not required.

Before continuing, we should mention that both `Gender` and `Region`, discriminating unions, are what is called **Sum types** in FP, which are itself a kind of **Algebraic Data Type** (or **ADT**), which we already encounter in the introduction. Sum types limit the option for a type to a list of exclusive choices. So we can only pick *one* choice. We are either children or adults, employed or unemployed. Boolean is an example of a sum type with two choices, true or false.

Sum types can be very useful, because they limit the number of values a type can have. Which means the compiler can more easily find mistakes. To model our domain, sum types are great because they give information about the domain in terms understandable by programmers and outsiders. A purchase we make could be either `ToPay` or `Paid`. Immediately, we know that for the first, there will be behavior (functions) for paying. The latter purchase would not need that kind of behavior, but would probably have things for tracking delivery or reimbursement.

Product types are another group of ADTs. Our incoming DTO is an example of a product type, where the possible choices are a product of their properties. They are useful and unavoidable. But

the number of possible combinations of values explodes rapidly. And the more possible values, the less our compiler can help us.

Instead of using sum types we could also have gone for a single type with boolean flags indicating the current 'state'. In which case we might have a `Purchase` type with a flag `isPaid`. This has several downsides. The type would be less extensible - what if we suddenly have a third option, called `UnPayable`? And how maintainable will our code be if we are passing two, three or more booleans (besides other info) to a function? Flags are also less informative than separate types, which tell you something additional about `Purchase` just by looking at the name.

Also, by accumulating several states/behavior in just one type, we will probably get a lot of information that is only relevant for one state. For example, a Purchase that has been paid for (isPaid is set to `true`) will need shipping information. Which we have to make optional because it is not a required part of an unpaid `Purchase`. So now a developer or business expert can add a shipping address to an unpaid purchase by mistake. And the compiler can no longer help us, because a shipping address is perfectly acceptable for the `Purchase` type. Which would not be the case if we had separate `ToPay` and `Paid` types. So in most situations, sum types are superior to (boolean) flags.

With the above type information avoiding mistakes has become easier. For instance, our user's gender can only be M, F or X. Our type system knows this and will use this information to keep us from making mistakes. Say our company really likes the patriarchy and wants to give beers exclusively to its male users. We might have a function like this somewhere in our code:

```
function canHaveABeer(user: User) {
    return user.gender === 'Male';
}
```

But it won't compile, telling us there is "*no overlap between Gender and the string 'Male'*". Which is true, because men are denoted with 'M' inside our domain. The compiler helped us avoid making a mistake. (Granted that it did not stop us from offending our entire female user base.) Similarly any billing or privacy logic is simplified by us having the right region already available.

We could go a bit further still. Two of our values are strings and we can easily envision situations where we have to pass both a first and last name to a function and accidently reverse the two. In addition, age is currently any possible number, but - assuming it's an age in years - in reality the age of our users will be an integer between `0` and about `150`. Which is way less than the maximum value of a number in Javascript.

So we will make things easier for other programmers and the compiler, with **newtypes**. Newtypes are used in Haskell to create a type based on exactly *one* field. So we can create a type based on a string or number - and we will only be able to use that type where it is appropriate. Which would not have been the case had we gone for a **type alias**.

```
type AltString = string; (1)

function printAlt(alt: AltString) {
    console.log(alt); (2)
}

const aString: string = 'a string'; (3)

printAlt(aString); (4)
```

(1) We create a type alias for string.
(2) We have a function that accepts parameters of type `AltString` and prints them.
(3) Here we create a string and set its type explicitly to string and *not* to the alias.
(4) And this parameter is perfectly acceptable to the compiler.

So with type aliases, we can make our code easier to read. But the compiler will not stop mistakes like confusing a first and last name. If the above had been a newtype, however, Typescript would have warned us that string is not assignable to `AltString`.

Newtypes are available as a separate package[8] for fp-ts, which you need to add to the dependencies if you want to use them. And how exactly would we use them for our `User`?

domain.ts

```
import {iso, Newtype} from "newtype-ts"; (1)
import {PositiveInteger} from "newtype-ts/lib/PositiveInteger"; (1)

// code for region and gender stays the same...

export interface FirstName
    extends Newtype<{ readonly FirstName: unique symbol }, string> {} (2)
export interface LastName
    extends Newtype<{ readonly LastName: unique symbol }, string> {} (2)

export const firstNameIso = iso<FirstName>(); (3)
export const lastNameIso = iso<LastName>(); (3)

export type User = {
    firstName: FirstName,
    lastName: LastName,
    age: PositiveInteger,
    gender: Gender,
    region: Region,
}; (4)
```

[8] https://www.npmjs.com/package/newtype-ts

(1) These are the imports we need from the newtype library. The first two are for creating our own newtypes, the third is a built-in type that restricts numbers to positive integers.
(2) Here we pick names and tell the library that these will be newtypes for string.
(3) These are not needed yet, but they will allow us to transform our newtypes into strings and vice versa.
(4) Now we start using both the two newtypes we defined and the built-in `PositiveInteger`.

Now if we try to pass an ordinary string to our `User` his name fields, it will pooh-pooh them. Only newtypes are good enough for our fancy type!

Maybe you are wondering about the performance implications of using newtypes. Well, in a language like Haskell, newtypes offer compile time safety without any runtime overhead because at runtime the newtype is simply the type it is wrapping! According to the fp-ts newtype library, performance for its implementation is almost the same as primitive types. So unless you have a very performance-sensitive application, you should be fine.

Validation with Either

So now we have types for our incoming data and users. Let's think about the functions we need. In this chapter, we are mostly focused on validation. So we need functions that:

- Make sure we have no empty or missing values. Everything is mandatory.
- Check whether we received a valid sex.
- Check the validity of age. Users for our kinky application have to be 18 or older. In order not to exclude any senior citizens, we will set the max to 150.

The one other thing we will have to do, is change our incoming values into those our User type is expecting. But let us start with validation.

domain.ts

```
import * as E from 'fp-ts/lib/Either'; (1)

export type FieldsNotEmpty = (e: UserRegistrationDto)
    => E.Either<string, UserRegistrationDto>;
export type ValidateAge = FieldsNotEmpty; (2)
export type ValidateGender = FieldsNotEmpty; (2)
```

(1) This style of importing is not needed for this particular example but it becomes handy when you are using more than one monad in a file, because there is an overlap in names (all monads have a map, chain, etc.).
(2) Since the signature of these two functions is the same as the above, we use a type alias.

You can import Either from the `lib` directory, or straight from the root of library: `import * as E from 'fp-ts/Either'`. The same goes for all the other monads we will encounter in upcoming chapters.

Every function will take the incoming event and perform one of the checks we mentioned earlier. The **Either** type that we are returning was not discussed in any detail yet. Either (alternatively called Result in, for example, F#) is a monad which is often used as an alternative for exception handling. It contains either (hence the name) a `left` or a `right` value. If the value is left and you call map or chain, *no further processing will occur*. Similar to Option's `none`, it just sits there and does nothing. The difference is that you can still retrieve the value that was originally put in there. Which is why Either is often used for error handling. If something goes wrong, stop processing by putting something (an error message preferably) in the `left`. If everything is fine and dandy, use `right` to do things with the value you have.

Note that our three functions, when successful, just pass along the `UserRegistrationDto` without changing it. This is quite common for validation functions, whose only job is to make sure the object that is passing through is valid. A function which takes a certain parameter and returns it unchanged is useful because it is easily composable. You can insert it in any chain between two functions as long as the output produced by the first is the type expected by your 'check'. One important side note is that they return an Either. So the caveat is that the functions in our composition have to handle monads.

Throwing code at you will make things clearer - right? Take a look at the implementation of the above functions. Again, these could be one-liners, but in book-form they are clearer when split over multiple lines. Note that in this chapter, we will put all our functions in a single file in the same directory (`src`) as our types, called `functions.ts`.

functions.ts

```
import * as E from 'fp-ts/lib/Either';
import { FieldsNotEmpty, ValidateAge, ValidateGender } from "./domain";

const fieldsNotEmpty: FieldsNotEmpty = (e) => {
    return e.firstName && e.lastName && e.age && e.sex && e.country ?
        E.right(e) : E.left('Not all required fields were filled in.');
};
const validateAge: ValidateAge = (e) => {
    return e.age >= 18 && e.age < 150 ?
        E.right(e) : E.left(`Received an invalid age of ${e.age}`);
};
const validateGender: ValidateGender = (e) => {
    return e.sex === 'M' || e.sex === 'F' || e.sex === 'X' ?
        E.right(e) : E.left(`Received an invalid sex ${e.sex}`);
};
```

Each function is short and simple. The first just checks if the fields are present. If they are, we pass back the event wrapped in an Either (`right` because everything is ok). If not, we put an appropriate error message in `left`. The other two have other, but very similar, checks. We could extract some common behavior, but that does not seem worthwhile yet.

Our application will have to use all these checks. So how do we combine them? There are several options. The first is perhaps the most simple. Apply one check and then use `chain` to apply the others one by one.

```
const exampleEvent: UserRegistrationDto = {
    firstName: 'Test',
    lastName: 'McTestFace',
    sex: 'M',
    age: 18,
    country: 'Belgium',
}

const first = fieldsNotEmpty(exampleEvent);
const second = E.chain(validateAge)(first)
const result = E.chain(validateGender)(second);
```

Remember, we have to use `chain` because we are getting back an Either. Without it nothing would be flattened and we would get `Either<string, Either<etc>>` monstrosities. Note that we are using point-free style again by only passing in the names of the functions.

The downside of this approach is that you are creating a lot of variables whose only job is to get passed to the next chain. Which means additional typing. We can do better with `pipe`:

```
import {pipe} from "fp-ts/lib/pipeable"; // additional import

const result = pipe(
    fieldsNotEmpty(exampleEvent),
    E.chain(validateAge),
    E.chain(validateGender),
);
```

This is cleaner. We do not have to explicitly pass stuff into the chains and we only need a variable for the result. The more maps and chains the cleaner this solution becomes when compared to the previous approach. Plus with piping the amount of explicit typing you need is limited compared to some other approaches. There are other options as well, but we will discuss those later. For now, note that the checks we defined can be used in a lot of different scenarios and are combined quite elegantly by a pipe.

It can be useful to break the above pipe and see what kind of type errors (if any) it starts spewing. What happens if you make `chain` return a string or a number? What happens if you return a `right` with a different kind of value inside?

Before we continue: Either or Validation?

If you are receiving data from a front-end where a user has just painstakingly filled in his information, it might not be ideal that the user needs to fix his errors one at a time, instead of just getting back a list of all problems. It might lead to user frustration plus additional calls to your backend because the data is resubmitted multiple times. Which is why **Validation** can be a useful alternative to Either.

To demonstrate its potential, we will rewrite our three checks. The first thing we are going to need is a **semigroup** for an array of strings which we will pass to the `getValidation` helper from Either.

```
import * as E from 'fp-ts/lib/Either';
import {getSemigroup} from 'fp-ts/lib/NonEmptyArray'

const applicativeValidation = E.getValidation(getSemigroup<string>());
```

This is the first time that we encounter a semigroup, so here is a brief explanation: a semigroup is a way of combining two values of the same type. That function is called `concat`. The product of two numbers, for example, is a semigroup where `concat` is the multiplication of the two number values. Sum is a semigroup where the numbers are combined by addition.

In our case, left/error is a string so we need a semigroup that can combine two strings. There are multiple options, like creating a new string from the concatenation of the two input strings. But we will change the signature of our functions and start using a string array as our left. Which is why we need a semigroup that can combine arrays. Luckily, there is a built-in method for that - which we just imported from the NonEmptyArray module!

```
import {NonEmptyArray} from 'fp-ts/lib/NonEmptyArray'; // additional import

type Validation = (e: UserRegistrationDto)
    => E.Either<NonEmptyArray<string>, UserRegistrationDto>; (1)

const fieldsNotEmptyV: Validation = (e) => {
    return e.firstName && e.lastName && e.age && e.sex && e.country ?
        E.right(e) : E.left(['Not all required fields were filled in.']);
}; (2)

const validateAgeV: Validation = (e) => {
```

```
    return e.age >= 18 && e.age < 150 ?
        E.right(e) : E.left([`Received an invalid age of ${e.age}`]);
}; (2)

const validateGenderV: Validation = (e) => {
    return e.sex === 'M' || e.sex === 'F' || e.sex === 'X' ?
        E.right(e) : E.left([`Received an invalid sex ${e.sex}`]);
}; (2)
```

(1) Create a type to save us some repetitive work.
(2) These functions are very similar to the ones we wrote previously, except we wrap our left in an array. This also means that instead of rewriting our code, we could have created a helper that wraps the string in an array. To do that, Either offers `mapLeft`, which only transforms left values. For clarity, we added a V suffix to the functions.

The pipe is very similar to our earlier one as well.

```
import * as E from 'fp-ts/lib/Either';
import {pipe} from "fp-ts/lib/pipeable";
import {sequenceT} from "fp-ts/lib/Apply"; (1)

const result = pipe(
    exampleEvent,
    e => sequenceT(applicativeValidation)(
        fieldsNotEmptyV(e),
        validateAgeV(e),
        validateGenderV(e),
    ), (2)
    E.map(([first]) => first), (3)
);
```

(1) `sequenceT` is for running multiple functions. The T stands for Tuple, because that is how we are passing them in. Alternatively, you can use `sequenceS`, where the S stands for struct, i.e. an object.
(2) We create a `sequenceT` which uses our `applicativeValidation` from before and receives our three checks.
(3) In case of success, every check will return a result (i.e. the `UserRegistrationDto`). So if everything goes well, we will have the same success value three times! So we use `map` to only take the first value of this array of identical DTOs, but picking the second or third elements would have worked equally well.

Now when we run this with an error in both gender and age, we get a response which contains all errors in one go!

```
{
  _tag: 'Left',
  left: [ 'Received an invalid age of 10', 'Received an invalid sex I' ]
}
```

We already mentioned why you should use validation. When should you prefer Either? Either has the advantage of being a bit simpler, so if you only have to do a few validations it might work equally well. Furthermore, maybe there is a check that takes a bit of calculating, say a synchronous password verification. If we can reject invalid requests before we have to run this expensive verification, we can save time and money. Which would not be the case with Validation.

Another reason to use Either, one that seems to be a lot less common, is when you have a few exclusive 'failure' modes that take widely different 'paths'. For example, when your success path returns a 2xx, but your failure paths are a 3xx, a 4xx or 5xx (i.e. redirects, bad requests and server errors). Remember: Either does not have to be used exclusively as an alternative for exception handling. In essence, it just stops further processing. So your choice will depend on your use case. We will be sticking to our earlier Either version of the validations for the remainder of this chapter.

Creating our user with lifting

Let us assume our data passed the above checks and is deemed valid. What now? Well, we should start building our User domain object. For the first and last name, this is pretty easy. Use the 'iso' we saw earlier and wrap our string in it. For gender we use one of various ways of asserting that the 'sex' string is a valid Gender since we (but not the compiler) now know it is ok. Age and country pose some difficulties. If we want to get a `PositiveInteger` for age, fp-ts will give us back an `Option`. Plus, we did not do any checks for the region. What if it happens to be invalid? Exploring this topic, we add the following functions to our domain:

domain.ts

```
import * as O from 'fp-ts/lib/Option';

export type CreateUser = (f: FirstName,
                          l: LastName,
                          a: PositiveInteger,
                          g: Gender,
                          r: Region) => User; (1)

export type FindRegion = (country: string) => O.Option<Region>; (2)

export type FindGender = (sex: string) => O.Option<Gender>; (2)
```

(1) We keep `CreateUser` clean. I.e. the incoming values are not Options that we have to check within the function. The only concern is creating a user from validated inputs. Creating an object in one go is recommended in Typescript.
(2) For `region` and `gender`, we play it safe and return the right types within an Optional value. Who knows what might go wrong? In addition, we will be using a built-in method for getting the positive integer, which will also return an Option.

We could have passed optionals into `CreateUser`, but that would mean a lot of dirty checks. So we want to avoid that if possible. And it is possible! But first, we have to discuss the concept of **lifting**.

Yes, there are other ways of handling the issue of creating the User. We could add checks to make sure the region is actually acceptable. And with validation we can pass multiple `right` values to a follow up function via map. But do remember, even after doing checks on external data, you might have some other calls you have to do that may or may not return data. So you may still have to deal with optionals. And yes, some `if/else` structures within the create might be acceptable in some cases. There are a lot of factors. But this is a book about FP and we are looking for nice functional ways to solve problems.

For the following, remember that monads are functors and get their map from Functor. So if you are unfamiliar with functors, just replace the word with monad in the explanation. Your understanding might be imperfect - but it will be good enough for this book.

According to Bartosz Milewski's lectures on category theory[9] the word lifting actually comes from the representation of the relation between a Functor and an ordinary mapping of functions as follows:

```
Fa ->  Fb   (3)

^       ^
|       |   (2)

a  ->   b   (1)
```

(1) This is the transformation of a value we call 'a' into a different value that we name 'b' represented by an arrow from left to right.
(2) The arrows upward signify a transformation of the value into a Functor, hence the F right before the character. We saw earlier that monads in many languages have an `of` function to put a value inside a monad. So the upward arrow would in practice be something like `Option.of(a)`.
(3) With this horizontal arrow we are dealing with transformations between Functors. A Functor with value 'a' is changed into a Functor containing a value 'b'.

[9] https://www.youtube.com/watch?v=oScqdMBk8Q8

All of this comes from category theory, where you deal with Objects and Morphisms. The former are the letters, the latter the arrows between them. So with `a -> b` the arrow is a morphism from object a to b. Once applied to FP, the objects are types and the arrows are functions.

So the horizontal arrows show functions transforming one value into another. And below it is just ordinary values we are dealing with, while above we see Functors wrapping those same values. Now lifting is simply moving from function (1) to function (3) and is called thus because of the convention of writing the version with the functors above the 'normal' one! We move 'up', into the realm of functors. Functional programming languages have specific functions for this. In Purescript these are called lift with a suffix indicating the number of arguments the lifted function takes. So `lift2` lifts a function of two arguments.

This we can use. We currently have a function that works with values, not functors or monads, i.e. `createUser`. But the functions that create the parameters we need produce monads. So we will lift our `createUser` function. And if functions that generate parameters do not return a monad, we'll wrap them in one.

A simple example in Purescript would look like this (ignoring the necessary imports):

```
firstValue :: Maybe String
firstValue = Just "a" (1)

secondValue :: Maybe String
secondValue = Just "b" (1)

needsLifting :: String -> String -> String
needsLifting a b = a <> b (2)

doesNotWork = needsLifting firstValue secondValue (3)

getLiftResult = lift2 needsLifting firstValue secondValue (4)
```

(1) Let us suppose these are the results from some other functions and they return `Maybe` values (which is the name for an Option in Purescript).
(2) This is a function that works with normal values, taking Strings and producing the concatenated version.
(3) Do this and your compiler will complain that it is expecting a String and getting a Maybe String, which are not the same. If you were to wrap it in one, the compiler would start complaining about the second parameter.
(4) But now we start with `lift2` and give it a function to lift AND the required number of parameters (i.e. two) which are already wrapped in a functor. And everything compiles. The function will return the result, wrapped in another Maybe - because the function has been lifted into the world of functors.

Back to Typescript, to put all this into practice. First, the easy part, writing an implementation for what we defined earlier, in a file called functions:

functions.ts

```
import {
    Region,
    Europe,
    NorthAmerica,
    Other,
    UserRegistrationDto,
    FieldsNotEmpty,
    ValidateAge,
    ValidateGender,
    CreateUser,
    FindRegion,
    FindGender,
} from "./domain";

const america: NorthAmerica = {
    _type: 'NorthAmerica',
};

const europe: Europe = {
    _type: 'Europe'
};
const other: Other = {
    _type: 'Other'
};

const countryMappings: Record<string, Region> = {
    Belgium: europe,
    USA: america,
    Germany: europe,
    China: other,
    // other countries
};

const createUser: CreateUser = (firstName, lastName, age, gender, region) => ({
    firstName, lastName, age, gender, region
}); (1)

const findRegion: FindRegion = (country) => {
```

```
    return countryMappings[country] ? O.some(countryMappings[country]) : O.none;
}; (2)

const findGender: FindGender = (sex) => {
    return sex === 'M' || sex === 'F' || sex === 'X' ? O.some(sex) : O.none;
}; (3)
```

(1) Just return an object with the given parameters as properties. Typescript is structurally typed, so this will compile because the 'structure' of what we are returning has enough overlap with the `User` Typescript is expecting.
(2) Use a Map (the data structure, not the function) which we just defined to find the correct country. If no value is found, return `none`.
(3) There are several ways of writing this piece of code. We do another validation which tells Typescript that the string is indeed of type `Gender`. If the string does not match one of the Gender values, we return `none`. Alternatively we could have gone for type assertions (`sex as Gender`) because at this point we are quite sure we have the right kind of string.

The recommended way of lifting in fp-ts is somewhat different from that in many other programming languages/frameworks. We are advised to use `sequence`, like we did for validation.

```
import {prismPositiveInteger} from "newtype-ts/lib/PositiveInteger";
import {sequenceT} from "fp-ts/lib/Apply"; (1)

const sequenceForOption = sequenceT(O.option); (2)

const user = pipe(
    exampleEvent,
    e => sequenceForOption(
        O.some(firstNameIso.wrap(e.firstName)),
        O.some(lastNameIso.wrap(e.lastName)),
        prismPositiveInteger.getOption(e.age),
        findGender(e.sex),
        findRegion(e.country),
    ),
    O.map(([f, l, a, g, c]) => createUser(f, l, a, g, c))
); (3)
```

(1) We need `sequenceT` for our lifting and `prismPositiveInteger` for getting the age as our newtype.
(2) We specialise the sequence for use with optional values.
(3) We are again using pipe to combine various actions. First we pass in our example event which is then passed to our `sequenceForOption`. In that sequence we call several functions. First and last names have to be transformed into the right newtype. Because they do not return an Option, we wrap them in one. The next three all return Options, so no wrapping is needed. Finally, we use `map` to pass the results of these functions to `createUser`.

When we call this `pipe` with a valid `UserRegistrationDto` we will get back a user within an Option. If one or more of the functions failed (and returned `none`) we get back an empty Option. The overall result is nice because all our functions are kept short, clean and understandable. You might now also understand why the monad 'constructors' (like `left`, `right`, `some` and many others) are called 'lifting functions'. I.e. they lift the given value into the monad. Note that `map` helps us lift functions that deal with ordinary values into the Monadic world as well.

In the functional world, `traverse` and `sequence` are used for handling 'lists' (by which I mean anything iterable, so not limited to a `List` type) of elements. Sequence can be used to 'swap' the list and monad. For example: going from a list of optionals to an optional list (but only if none of the optionals contained empty values). Traverse is similar but is also used to apply a function to each element in the 'list'.

Bringing it together

Ok, so now let us fit together all the pieces of the puzzle. Functions, assemble! The one thing we should add, is some kind of unified response. We will use this http/rest inspired type:

```
type Response = {
    status: number,
    message: string,
};
```

Where the status element will be a 2xx, 4xx or 5xx number signifying success or failure (so a http response code) with the message giving additional information. We will also add some pure helper functions for creating responses.

functions.ts

```
const createdResponse = (message: string): Response => ({
    status: 201,
    message,
});

const badRequest = (exception: string): Response => ({
    status: 400,
    message: exception,
});

const internalServerError = (): Response => ({
    status: 500,
    message: 'failed to create',
});
```

There is some duplication here that we could factor out, but this is good enough. Now brace yourselves for almost 30 lines of code!

```
function userResponse(u: UserRegistrationDto) {
    return pipe(
        u,
        e => sequenceForOption(
            O.some(firstNameIso.wrap(e.firstName)),
            O.some(lastNameIso.wrap(e.lastName)),
            prismPositiveInteger.getOption(e.age),
            findGender(e.sex),
            findRegion(e.country),
        ),
        O.map(([f, l, a, g, c]) => createUser(f, l, a, g, c)),
        O.map(u => {
            console.log(`Created ${JSON.stringify(u)}. Could now save in db`);
            return createdResponse(`Created ${JSON.stringify(u)}`);
        }),
        O.getOrElse(internalServerError),
    );
} (1)

function flow(u: UserRegistrationDto) {
    return pipe(
        fieldsNotEmpty(u),
        E.chain(validateAge),
        E.chain(validateGender),
        E.map(userResponse),
        E.getOrElse(badRequest),
    );
} (2)
```

(1) This function is similar to the one we presented in the previous chapter, in that it builds our user. The only difference is that after doing so, it logs a message, which is a stand-in for a call to a database, and builds a `Response`. This is either a 201 ('created') in case of success, or a 500 in case of a failure (we assume our validation will stop any errors caused by the user, so a 5xx seems appropriate). For the latter, we use `getOrElse`, which will call the given fallback function if it is passed a `none`. So this function either assembles a user and logs, or it becomes empty at some point and goes to the fallback.
(2) This is our entire (work)flow brought together. First we do our three checks. Next, we use the function we defined at (1) to create the user and get a response. At the end we either retrieve the answer from that function or we return a 400 ('bad request') if our validation failed. For that we use Either's `getOrElse` which is similar to that of Option.

For a valid example event, something like this will be returned:

```
{
  status: 201,
  message: 'Created {"firstName":"Test" ...}'
}
```

But if we change the age to 10 - too low - we get:

```
{
    status: 400,
    message: 'Received an invalid age of 10'
}
```

All in all, the 'application' consists of some 50 lines of type definitions and almost a 100 lines of actual code, with the double caveat that we were not really trying to be as brief as possible (making some of the functions one-liners could shorten it by 20-40 lines) *but* Javascript is quite the succinct language. 50 lines, that's about the length of one medium Java class with all its getters, setters and other methods - at least in the days before Lombok (in which case 50 lines is about the length of the class generated by Lombok).

If we look through our code again, we can see the broad outlines of our program. Our types take up quite a lot of space - making our program about 50% longer than the native Javascript version. But they help. While writing the above snippets that bring everything together, type errors showed me where I was making mistakes. And when everything compiled, I was almost 100% certain that the code would actually work. Yes, that certainty is never a hundred percent, definitely not when you are working on a bigger, more complex application. But the types are clearly worth the additional effort.

The functions themselves are pretty neat as well. Most of them are very short (a few lines, often reducible to just one) and pure. We also avoided exceptions by using monads. Which means writing tests is easy (not a lot of possible paths in a one-liner) and combining is a breeze as well. The last two, bigger functions are proof of that. With only a little effort we end with something that reads as a linear story: "Ok, so flow is the entry point. We do a few checks first, create a user - and an appropriate response. And in case of failure, we'll end up here in the getOrElse and return something fitting." How is that for maintainability?

Btw, you may have noticed that we are using pipe() a lot *and* I already mentioned how it seems to be the default way of handling transformations in fp-ts. So should you use pipe everywhere? No, but it might be a good default *and* it helps fp-ts with type inference! Piping from one function to the next keeps ceremony minimal (no unnecessary creation of variables, passing of parameters etc.), is readable and easy to reason about. But you might run into situations that don't fit as well or where another approach would give you more maintainable code. In which case, go for whatever alternative seems more appropriate!

Change is inevitable - how to cope with it

Well this is all fine and dandy, writing a greenfield application in FP. But often, that's the easy part. Once the application is running, it will have to evolve alongside changing business requirements. How will our code cope with change? Let's try making a few changes and see what happens, while taking care not to introduce too many new concepts.

First, an easy one: an additional 'check'. Since we haven't done any checks on `country`, we're going to refuse users from Germany. After all, as Belgians we have the right to hold a bit of a grudge against a nation that invaded us twice in a single century.

The function type will be similar to that of the other checks.

domain.ts

```
export type ValidateNotGerman = FieldsNotEmpty;
```

And the code might look like this:

functions.ts

```
const validateNotGerman: ValidateNotGerman = (e) => {
    return e.country !== 'Germany' ?
        E.right(e) : E.left("We don't like your kind around here");
};
```

Not that hard. To add it to our application, just put the check behind the existing ones.

functions.ts

```
function flow(u: UserRegistrationDto) {
    return pipe(
        fieldsNotEmpty(u),
        E.chain(validateAge),
        E.chain(validateGender),
        E.chain(validateNotGerman), // this is the only change
        E.map(userResponse),
        E.getOrElse(badRequest),
    );
}
```

The application gives back the same results as before, except when we pass in Germany as a country.

```
{
    status: 400,
    message: "We don't like your kind around here"
}
```

That was easy right? We created a new function and added it to our pipe after our other checks. Perhaps most importantly, from the viewpoint of extending an existing application, *we did not touch any existing functions* (excluding `flow` - which still has the same signature though). We did not change any of the checks that were already present. Most of the time, this is a good idea. We know the existing code is stable. It has proven its worth in production and it probably has a lot of tests. And if we were to change existing code, we risk introducing bugs and doing a lot of additional work because unit tests have to change. Instead, we write additional code and 'insert' it into our application as if there was a slot available. Sure, a few tests for our entire application might start failing, but that is expected - even desired - behavior. But all our other code is safe and sound.

How about a change to the `User` domain object? What if our business had become woke and decided to give preferential treatment to everyone but men, in recompense for its past ways? What if it decided to add a `customerStatus` field, setting the other genders to VIP? One possible place to add this, would be in `createUser`. You can probably guess how we would go about doing this:

- Add customer status to `User`
- Create an additional function `findStatus`
- Add an additional parameter to the `createUser`
- Bring everything together

And that will work. On the other hand, maybe our `createUser` is old and has a lot of tests depending on it. So maybe we would like to leave as much of the current code untouched. If that is the plan, there are alternatives, like this one:

domain.ts

```
export type CustomerType = 'Normal' | 'VIP'; (1)

export type User = {
    firstName: FirstName,
    lastName: LastName,
    age: PositiveInteger,
    gender: Gender,
    region: Region,
    customerType?: CustomerType, (2)
};
```

(1) As always, we want to avoid a simple string because that has sooooooooooo many possible values. So we add another type, limited to the two possible statuses within our system.
(2) We add this `customerType` to our User as an *optional* field, hence the question mark.

This way, we do not break any existing code. An optional field was added? Whatever, Typescript does not care. Now we write this brief function:

functions.ts

```
const addStatus = (u: User): User => ({
    ...u,
    customerType: u.gender !== 'M' ? 'VIP' : 'Normal',
});
```

Which creates a new `User` and adds the customer type. The rest operator (...) is used to add all existing properties/values. Again, thanks to nice types, you won't be able to compile this if you mistype 'M' as 'N' or 'Normal' as 'Bormal'.

Alternatively, we could have used a lens from monocle-ts[10] to change the field. As we mentioned before, lenses are pure getters and setters that make it easy to change or retrieve a property from a structure.

```
import {Lens} from 'monocle-ts'
import {User} from "./domain";

const customerTypeLens = Lens.fromProp<User>()('customerType'); (1)

const addStatus = (u: User) => {
    const modified = customerTypeLens
        .modify(c => c ? c : u.gender !== 'M' ? 'VIP' : 'Normal');
    return modified(u);
}; (2)
```

(1) We create a `User` lens for the `customerType` field. The compiler will warn you if you pass in a field that does not exist.
(2) If `customerType` is not filled, we set it using `modify` and apply our resulting function to the incoming `User`. We return the result, which is a `User` with one of its fields modified.

This becomes a lot more useful when you have nested structures, like in this example:

[10] https://github.com/gcanti/monocle-ts

```
import {Lens} from 'monocle-ts'
import {CustomerType, FirstName, Gender, LastName, Region} from "./domain";
import {PositiveInteger} from "newtype-ts/lib/PositiveInteger";

type AlternativeUser = {
    firstName: FirstName,
    lastName: LastName,
    gender: Gender,
    additionalInfo: {
        personalInfo: {
            age: PositiveInteger,
            region: Region,
        },
        otherInfo: {
            customerType?: CustomerType,
        }
    }
}; (1)

const customerTypeLensAlt = Lens.fromPath<AlternativeUser>()(
    ['additionalInfo', 'otherInfo', 'customerType']); (2)

const addStatusAlt = (u: AlternativeUser) => {
    const modified = customerTypeLensAlt
        .modify(c => c ? c : u.gender !== 'M' ? 'VIP' : 'Normal');
    return modified(u);
}; (3)
```

(1) This time we have a slightly different user type with a nested `customerType` (under `additionalInfo/otherInfo`).
(2) We create a lens, passing in the entire path as an array of strings. Alternatively, you can write three lenses (one for each field) and use `compose` to combine them.
(3) This is the same as before, with the same kind of result: the `customerType` is changed.

Lenses are a useful way of setting and retrieving properties within more complex objects, but for our use case they might be overkill. So we will stick to our first approach and add the original `addStatus` to our program:

functions.ts

```
function userResponse(u: UserRegistrationDto) {
    // almost everything is the same as before
    return pipe(
        u,
        e => sequenceForOption(
            O.some(firstNameIso.wrap(e.firstName)),
            O.some(lastNameIso.wrap(e.lastName)),
            prismPositiveInteger.getOption(e.age),
            findGender(e.sex),
            findRegion(e.country),
        ),
        O.map(([f, l, a, g, c]) => createUser(f, l, a, g, c)),
        O.map(addStatus), // except for this line!
        O.map(u => {
            console.log(`Created ${JSON.stringify(u)}. Could now save in db`);
            return createdResponse(`Created ${JSON.stringify(u)}`);
        }),
        O.getOrElse(internalServerError),
    );
}
```

And the customer type is added to the user! Very little existing code was changed in order to make this work. But the approach does have its downsides. What I find most annoying is that we added an optional field which might make using `User` more difficult. So another approach would be to change the existing type to something like `BasicUser` or `UnvalidatedUser`, subsequently re-introducing the `User` type - but this time with a required customer type:

domain.ts

```
// this is equal to our previous User type
export type BasicUser = {
    firstName: FirstName,
    lastName: LastName,
    age: PositiveInteger,
    gender: Gender,
    region: Region,
};

// this is our 'real' user which we'll be using throughout our domain
export type User = {
    firstName: FirstName,
    lastName: LastName,
    age: PositiveInteger,
```

```
    gender: Gender,
    region: Region,
    customerType: CustomerType,
};
```

Now we let createUser return `BasicUser` (the Typescript compiler will already be complaining about its return) and make `addStatus` return the real user. Ideally, we would have discovered at least part of this complexity before working on our implementation, through discussion with the domain experts. If they had told us there were a lot of validation steps - and that they might change - we could have added an 'Unvalidated User' type beforehand. But however great our interaction with the domain experts, some missing requirements are to be expected.

This approach works AND has little impact on the existing code. `createUser` is slightly impacted by the change of its return type and a change of name would be in order as well. But that is the kind of refactor you can do automatically with any modern IDE. After that, once we add a function (like `addStatus`) to our pipeline that takes a basic/unvalidated user and returns the real deal, everything will compile again. Functions further 'downstream' in our flow remain the same and keep working with the User type. So if you think more of these special rules are incoming, you may prefer this approach above the earlier ones. Apart from producing an additional type, it has many upsides.

While you may not believe the general case yet, I hope you've seen that extending this particular application with additional rules, validation, etc. is not hard. Within the lego-like structure of FP, it is easy to add an additional piece without interfering with what is already present.

Summary

In this chapter, we wrote our first application with fp-ts. Along the way, we had another encounter with the Option Monad and learned how to use the Either Monad. We saw how to combine small functions into a larger whole and talked about how the killer combo FP + types can make our code simple to reason about and easy to change. We also learned how to add types where we needed them, how type-safety can make our applications better and that our app is quite easy to extend with additional features.

Of course this is not the end of our journey - it is in fact just the beginning. In the next chapters, we will write more FP applications and learn how we can build them, test them, expand them.

Deploying a monitoring application

In our first project we consciously limited ourselves to the basics. We created an example event and passed it to our application from within our IDE. Furthermore, results were simple printed to output instead of saving to a database and taking other additional actions.

Now, we are going to expand our scope. This time, we will actually deploy and run our application in the cloud, getting us one step closer to what a 'real' project looks like. And instead of starting with our types, this time we will go for a more loose approach, coding as we go. But first, a word about our platform: AWS Lambda.

AWS Lambda and SAM

For those few who are unfamiliar with Amazon Web Services (AWS), it is the world's largest cloud provider. Instead of running your software on your own computer or in a data center, you pay AWS to run what you want using their compute services, like the most basic one, EC2. In addition to pure compute, AWS also offers storage (S3), databases (both SQL and noSQL), containerized compute (ECS, EKS, Fargate), AI and a lot of other useful services (like SQS for queues or SES for emails). Much of what is offered falls under the category of managed services, where the cloud provider takes care of all/most of the underlying infrastructure for you.

Many, many IT years ago - some six human years - AWS introduced 'serverless' compute with AWS Lambda. Lambda allows you to run code without worrying about infrastructure. You tell it what to run, when to run, what memory the 'function' requires (things like CPU are automatically configured based on how much memory you select) and how long to run without timing out. Meaning if a job takes longer than the limit you set, it will be stopped. Combined with the wide variety of AWS managed services, Lambda allows developers to create many kinds of applications without managing any underlying hardware and software. There are still use-cases where you are better of not using Lambda, but for many who have worked with serverless, it has become the default. I.e.: use Lambda *unless there is a really good reason not to.*

The fact that Lambdas are also called functions might confuse the reader. Overloaded terminology ahead! I hope context will help differentiate functions in FP from those running on AWS.

So why use Lambda for this book? Several reasons. First, I have a lot of personal and professional experience with AWS' serverless. Which means it is a small effort for me to add code for building and deploying. Second, it also simplifies matters for the reader. No need to set up infrastructure, perhaps via a console and with ssh'ing into an instance, and making sure your environment is identical

to mine. Just deploy the project with bash - or manually - and everything will work. Third, for poc'ing or exploring, Lambda is dirt cheap. I've done a lot of experiments with it and have several functions running in my personal account every day. I have never had to pay a cent for all that compute. Compare that to countless warnings from online courses reminding you to turn off your (ec2/ecs/eks) instances lest you get an unexpected cloud bill!

Finally, I think AWS Lambda and serverless in general are a good fit for FP. Both like to abstract away concerns, in the case of FP low-level concepts like loops, in that of Lambda the underlying infrastructure. Both like to create small components (functions) that can be combined - in the case of AWS with other managed services - to create larger applications. FP tries to keep most of its code/functions pure, AWS Lambda is stateless. Etc.

There are several options for creating applications in the AWS Cloud. For experimenting, you can just click together what you need in the console. Or you can use the CLI/SDK and create your application via code. But for real applications, Infrastructure as Code is recommended. Infrastructure as Code (or IaC) is describing your infrastructure at a high level in for example a text document. You define *what* you need without going into the *how*. You leave it up to your cloud provider to actually bring your description to life.

This has many advantages over clicking together an infrastructure or the imperative commands of a CLI. It means you know exactly what kind of infrastructure you have, because it is described in the document you created, which servers as your **single source of truth**. And your information about what your infrastructure really looks like does not turn stale as long as you only change infrastructure by changing that document. It is also really easy to recreate your setup (just delete the current infrastructure and run the document again), or move it to a different AWS 'region'.

In AWS, IaC is provided by AWS CloudFormation. With CloudFormation, you describe your infrastructure as a JSON or YAML file called a *template*. The various pieces of infrastructure you need (like an S3 bucket, SQS queue, EC2 instance or Lambda) are called *Resources*. Once you send your file to AWS and your actual infrastructure is created, the result - shown in the CloudFormation console - is called a *stack*. So instead of telling you what to click in the console, we are going to write a template that will automatically create the infrastructure stack for this chapter.

More specifically, we will be using the Serverless Application Model (SAM), which is a layer on top of CloudFormation specifically designed to simplify the creation of serverless applications. There are several interesting alternatives, including the AWS CDK (definitely more appropriate for complicated or non-serverless projects), Troposphere, Terraform and the powerful Serverless framework. Most of these also created Cloudformation 'under the hood'.

If you do want to follow along and deploy this application in the cloud, you will have to set up a few things. You need to create an AWS account if you do not have one yet. You also have to install the AWS CLI[11] and set default credentials[12] with either the right permissions for deploying, or Admin - with the latter certainly recommended if you are an AWS newbie. You will also need an S3 bucket to store the zipped code in. You can create one manually via the AWS console if you do not have one yet. If you don't want to do any of this, that's fine as well - just follow along with the code.

Setting up our project

The idea of this chapter is to *create an application that will monitor the metrics of a single Lambda*. Of course monitoring multiple Lambdas, or perhaps all your AWS resources, would be a lot more useful - but we have to limit the scope of our project, so instead I will suggest ways to expand the reach of this application at the end of the chapter. The source of our metrics will be Cloudwatch, which is the logging and metrics service of AWS. Lambdas automatically send logs and metric info there.

First we will set up the project and infrastructure with a dummy Lambda. Next, we will write functional code that retrieves metric information from Cloudwatch regarding 'invocations' (how many times was the Lambda function called?), 'errors' (how many errors did it produce?) and 'throttles' (you can be throttled by AWS when you run into certain limits). Depending on the information we retrieve, we will return a status with value 'ok', 'warning' or 'error', and add a string with some additional information.

To make things easier, we will be monitoring… the function we are creating. Inception! Pretty useless, sure, but it also means that readers do not have to set up some mock Lambda in their account if they do not have one running yet.

We will start by setting up our environment and infrastructure, getting all of that out of the way before we start writing our actual functional code. Create a new project via your favorite IDE. If a package.json was not added automatically, add it now. It should contain something like the following.

package.json

[11] https://docs.aws.amazon.com/cli/latest/userguide/install-cliv2.html
[12] https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-quickstart.html

```
{
  "name": "leanpub-monitoring",
  "version": "1.0.0",
  "description": "Example monitoring project with fp-ts and aws lambda",
  "main": "index.js",
  "scripts": {
    "clean": "rm -rf node_modules && rm -rf dist",
    "zip": "cd dist && zip -r cloudwatch.zip *",
    "build": "npm i && webpack && npm run zip"
  },
  "dependencies": {
    "fp-ts": "^2.8.3"
  },
  "devDependencies": {
    "aws-sdk": "^2.764.0",
    "typescript": "^4.0.3",
    "ts-loader": "^5.3.3",
    "webpack": "^5.3.0",
    "webpack-cli": "^4.1.0"
  },
  "author": "Sam Van Overmeire",
  "license": "ISC"
}
```

For now our only `dependency` is fp-ts. We also have a couple of `devDependencies`. Typescript and all types are typically added as `devDependencies` because they are not needed at runtime. The AWS (Node) SDK is available within the AWS Lambda runtime environment so we do not have to add it. This is useful, because it means the zip file (containing the code we want to run on AWS) which we will be creating will be smaller. And a smaller zip means our Lambda will start faster (downloading, unzipping and setting up code takes time). Adding the SDK as a normal dependency does have the advantage of making sure the version you use locally is the same as that running in your actual application, which avoids mismatches. We also add webpack, webpack-cli and ts-loader. Webpack will be used to transform our Typescript code into a single Javascript file, placed in a folder called `dist`. We will zip that file, so we can later upload it to AWS. All of this happens in the build script, using some commands we defined under `scripts` (namely `build` and `zip`).

With webpack the difference between normal and dev dependencies is less important, because webpack will throw out stuff that isn't used (**tree shaking**). But the extra effort in putting these dependencies in the right location is very small and might be useful if we decide to move away from webpack.

In the root of your project, you will also need a tsconfig containing something similar to the

following. Most of the specifics are not very important for running the project, though without a recent version of Ecmascript (configured in the `lib` section), the project will not compile.

tsconfig.json

```
{
  "compilerOptions": {
    "lib": [
      "es2017",
      "esnext"
    ],
    "esModuleInterop": true,
    "moduleResolution": "node",
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "sourceMap": true,
    "target": "es2017",
    "outDir": "dist",
    "noImplicitThis": true,
    "noImplicitAny": true,
    "strictBindCallApply": true
  },
  "exclude": [
    "node_modules",
    "__tests__",
    "dist"
  ]
}
```

We also need to tell webpack to output to our dist folder, to use ts-loader (so that it will correctly handle Typescript code) and to ignore the aws-sdk since it will be provided externally.

webpack.config.js

```
const path = require("path");

module.exports = {
    resolve: {
        extensions: [".js", ".jsx", ".json", ".ts", ".tsx"],
    },
    output: {
        libraryTarget: "commonjs",
        path: path.join(__dirname, "dist"),
        filename: "[name].js",
```

```
    },
    target: "node",
    externals: ["aws-sdk"],
    module: {
        rules: [{ test: /\.tsx?$/, loader: "ts-loader" }],
    },
};
```

Now create an `src` directory, as a home for our Typescript files. In that directory, create an index file containing the simplest possible Lambda: one that just returns a "hello world" whenever it is triggered.

index.ts

```
export const handler = async () => {
    return "hello world";
};
```

Now all that remains is the infrastructure file, which we will place in the root of the project. (As always, ignoring the numbers between brackets, which denote remarks from me.)

infrastructure.yaml

```
AWSTemplateFormatVersion: 2010-09-09 (1)
Transform: AWS::Serverless-2016-10-31 (2)

Resources:
  MetricsLambda: (3)
    Type: AWS::Serverless::Function (4)
    Properties:
      Handler: main.handler (5)
      Runtime: nodejs12.x (6)
      MemorySize: 512 (7)
      Timeout: 20 (8)
      CodeUri: dist/cloudwatch.zip (9)
      Policies: (10)
        - AWSLambdaExecute
        - Version: "2012-10-17"
          Statement:
            - Effect: Allow
              Action:
                - "cloudwatch:Get*"
                - "cloudwatch:List*"
              Resource:
                - "*"
```

Well, this calls for some explaining. Here we go.
(1) This is a required header. It tells AWS that this is a CloudFormation template, describing a collection of resources.
(2) Because we are not using vanilla CloudFormation, we tell AWS that we are using SAM on top of it. Else our special types will go unrecognized.
(3) Under the `Resources` header, we define the resources we need, in this case just a Lambda. `MetricsLambda` is the name we picked for it. When deployed in your account, AWS will generate a unique name, consisting of the name you select for your 'stack' followed by this name and a random suffix.
(4) The type tells AWS what kind of resource it has to create. `AWS::Serverless::Function` is a type defined by SAM. It is basically the Cloudformation type `AWS::Lambda::Function` but with added conveniences.
(5) The `handler` is the 'entrypoint' of a Lambda application. When the Lambda starts, it will look for the given entrypoint and start running the function it finds. Here we tell AWS that inside the zip there is a file called `main.js`, which contains a function called `handler`. Which we defined just before we started writing this template. So on start up, AWS will open the file and call our function - hopefully returning "hello world" to us.
(6) This tells AWS that our Lambda requires Node/Javascript.
(7) A memory size in megabytes. The minimum is 128. As we mentioned previously, other aspects of the Lambda like the amount of CPU are based on this value. 512 should be more than enough for us, since we will be mostly waiting for data from Cloudwatch calls to come back to us, after which we will do some minor transformations.
(8) The maximum amount of time our function will run, in seconds (the minimum is 1). 20 should be more than enough for gathering up a few metrics. But it is better to estimate a bit too high than a bit too low - because in the latter case, your Lambda will never return a result and you might go crazy trying to figure out what is wrong with your code. Setting it way too high is not a good idea though. If your Lambda gets 'stuck' and only stops when reaching that high timeout limit, your costs might become a lot higher than estimated.
(9) The location of our code inside our project. `npm run build` will generate a zip inside the `dist` folder called "cloudwatch.zip", so that is where we point to.
(10) Policies are part of AWS's Identity and Access Management (IAM). They tell AWS what the Lambda is allowed to do within our account. Because we want to gather metrics, we give it read permissions on Cloudwatch. `AWSLambdaExecute` is a policy managed by AWS, which allows a Lambda some basic functionality, like writing logs and metrics to Cloudwatch.

Now the finishing touch for our setup, a small bash script.

setup.sh

```bash
#!/bin/bash

set -eo pipefail

STACKNAME="monitoring-example" # or any other name you like

if [ $# -lt 1 ]; then
    echo "Needs a bucket!"
    echo "Usage ./setup.sh <bucket>"
    exit 1
fi

BUCKET=$1

echo "Running build"
npm run build

echo "Deploying stack"

aws cloudformation package \
--template-file infrastructure.yaml \
--s3-bucket "${BUCKET}" \
--output-template-file outputSamTemplate.yaml

aws cloudformation deploy \
--template-file outputSamTemplate.yaml \
--stack-name ${STACKNAME} --capabilities CAPABILITY_IAM

echo "Removing temporary SAM file"
rm outputSamTemplate.yaml
```

The script requires an S3 Bucket as an argument because we will store our zip files in that bucket. If the script gets an argument for `BUCKET`, it will first build our project, creating the zip of our code using webpack. Once that is done, we use the `cloudformation package` command from the AWS CLI to generate a CloudFormation template from our SAM template. This command will also look for the CodeUri we specified earlier, upload the zip to S3 and will refer to that uploaded file in the template. Once that is done, all that remains is to call `cloudformation deploy` with the generated template. We also pick a name for the stack and tell AWS that it is ok if it has to generate an IAM policy (which Lambda will require to talk to Cloudwatch, see the `Policies` section of the infrastructure file).

If you want to run this, make sure you have the cli installed & default credentials configured (see the requirements earlier in this chapter). Also, to run this bash file, you will have to set it to executable (for which you can use `chmod` in a unix environment).

If all that is done, we're good to go and you can run this in your terminal (with the real name of your S3 bucket instead of the placeholder):

```
./setup.sh THE_NAME_OF_YOUR_BUCKET
```

Once the script has run, the end result is a Lambda, which you can run manually if you want to verify that everything works. You can do this via the AWS Console[13] or the AWS CLI. The function should return "hello world". If this is correct, then you are all set to go. And you have just made sure that your Lambda has been invoked, so you already have some metrics to query!

Using the Cloudwatch SDK

The most important call for our project will be `getMetricStatistics` from the AWS Node SDK, which "*Gets statistics for the specified metric*". Given the right parameters, we can retrieve our three relevant metrics for a function using this call. So we might first write one big function for retrieving the metric information.

```
// necessary imports

const getMetrics = (functionName: string, start: Date, end: Date) => {
    const params = {
        // build parameters here
    };

    cloudwatch.getMetricStatistics(params).promise() (1)
        .then(data => {
            // do stuff with data
        })
        .catch(err => {
            // do stuff with the error
        });
} (2)
```

(1) By default the AWS SDK uses callbacks, but if you add `promise()` it will use promises instead (in v3 promises have become the default). Promises are often better than callbacks and in our case they will also play nice with some of the fp-ts async stuff we will be using.
(2) We are using arrow style for this function, as we will do a lot in the upcoming pages.

And this is fine for writing a POC. But it has some downsides - and maybe you can already guess a few of those. The most important one is that, because we are doing I/O, this function is impure.

[13] For the eu-west-1 region (Ireland), the link to the Lambda console is https://eu-west-1.console.aws.amazon.com/lambda/home?region=eu-west-1#/functions

Furthermore, there are a lot of pure parts within the function, which we could have extracted with little effort, reducing the function with the AWS SDK call to the bare minimum. Splitting up this function in pure and impure parts would have some additional benefits, like making every individual part easier to understand, test and reuse.

So, a slightly different approach is warranted. Under `src`, create a `metric-gateway` folder. The gateway signifies that this folder contains calls to other, external systems. In this case it is just metrics, so we add that to the name. Under this folder, add a Typescript file with the following content:

calls.ts

```
import CloudWatch, {GetMetricStatisticsInput, GetMetricStatisticsOutput}
    from "aws-sdk/clients/cloudwatch";

const cloudwatch = new CloudWatch({ region: 'eu-west-1'}); (1)

export const getCWMetrics = (params: GetMetricStatisticsInput):
Promise<GetMetricStatisticsOutput> => {
    return cloudwatch.getMetricStatistics(params).promise(); (2)
};
```

(1) Set up the SDK's Cloudwatch client which we will need for calling that AWS service. Note that the import goes straight to the correct client. You can also get a client by going to the root aws-sdk package, but this will add some (unneeded) initialization logic and will be slightly slower. As config we pass in the `region` we are working in, which in my case is eu-west-1 (Ireland). Make sure this matches the region you deployed your stack in, else you will get an error that the function you are trying to retrieve does not exist.
(2) The basic call is the same as before, except it no longer contains the pure logic for creating the parameters or handling what we get back. Unlike the example we gave before, we have added (not strictly needed) types to the input and return.

In the same folder, create a file with the following (*note that this will not compile yet*):

params.ts

```
export const generateCWMetricStatisticsParams =
    (metricName: Metric, between: Between) => (fun: string) => {
    return {
        StartTime: between.startTime,
        EndTime: between.endTime,
        MetricName: metricName,
        Namespace: 'AWS/Lambda',
        Period: THREE_HOURS_IN_SECONDS,
        Dimensions: [
```

```
            {
                Name: 'FunctionName',
                Value: fun,
            },
        ],
        Statistics: ['Sum'],
    };
};
```

Here we create the parameters we need for the Cloudwatch call. The easiest arguments are `StartTime` and `EndTime`, which are dates. Because these belong together, we create a type `Between` which serves as a container for both values. `MetricName` is also pretty self-explanatory. To pass it in we also create a specific type, `Metric`.

`Namespace` tells Cloudwatch where it will find the metrics. Lambda will automatically put its metrics under the 'AWS/Lambda' namespace. We can retrieve metrics in various ways. Since we want the metrics for a function, we use the function name to only get relevant values.

`Period` tells AWS how detailed you want the information to be. Do you want to see the metric statistics for every second? Minute? Hour? We pick a high period because we are not interested in any details. We only want to know if there were any errors/throttles/invokes at all, not at what precise moment they occurred. For similar reasons, we only require the `Sum` statistic.

Now let's get this to compile. Under `src`, we will add a folder called `util`. This will contain a types file:

types.ts

```
export type Between = {
    readonly startTime: Date,
    readonly endTime: Date,
}

export type Metric = (typeof METRICS)[number]
```

Here we define the `Between` type we met earlier. Note that we added `readonly` to both dates, which is something we already added to some fields in the previous chapter. It is a compile time annotation that Typescript will use when checking your code. If it sees you mutating fields with this annotation, it will throw a warning. From now one we will try to add it to every property we define because, as good FP practitioners, we want to avoid mutability wherever we can.

Note that, once again, our code will not compile. So sad. Why? Because `METRICS` is undefined. Uh. Fine. Create another file in the `util` folder.

constants.ts

```
export const METRICS = [
    'Invocations',
    'Errors',
    'Throttles',
] as const; (1)

export const THREE_HOURS_IN_SECONDS = 10800; (2)
```

(1) Note that this `as const` is required if we want to create a type from this array.
(2) Ow yeah we used this constant in our parameter-generator-function (**params.ts**). So better define it while we're at it.

The first `const` contains our three metrics, which will be useful later on. In `types.ts`, we use a trick to create `Metric` from those three values. The second is pretty self-explanatory. Now go back to the previous files and fix the imports by hand or with some help from your IDE. Everything should now compile. We have now created functions for getting our metrics. But to make life easy for users of our metric-gateway, we will create an 'entrypoint', which will contain an easy to use call for all the asked-for metrics of a specific function.

First, though, create another folder under `src`, called `transformations`. This will contain the pure transformations that we will do on the data we get from Cloudwatch. Create a file in that folder, creatively called transformations.

transformations.ts

```
import {Metric} from "../util/types";
import {GetMetricStatisticsOutput} from "aws-sdk/clients/cloudwatch";

export const metricsToStatsResult = (functionName: string, metricName: Metric) =>
    (d: GetMetricStatisticsOutput): StatsResult => {
        return {functionName, metric: metricName, data: d.Datapoints}
    };
```

Currently, it only contains one function, which knows how to take a function, metric and Cloudwatch output (if you look back at your calls file, you will see that that function returns a Promise containing `GetMetricStatisticsOutput`) and combine those into something useful. Note that functions in this file will not use any Monads from fp-ts. We do not want to use Monads when they are not necessary and for pure transformations we only need regular values.

Our function returns a `StatsResult`, which is a new type.

types.ts

```
export type StatsResult = {
    readonly functionName: string,
    readonly metric: Metric,
    readonly data: Datapoints,
}
```

Again, everything is `readonly` because we loath mutation. The `Datapoints` type, btw, is defined by AWS and is just an alias for an array of numbers. Now we can create our entrypoint:

entrypoint.ts

```
import * as TE from "fp-ts/lib/TaskEither";
import * as A from "fp-ts/Array";
import {Between, Metric, StatsResult} from "../util/types";
import {generateCWMetricStatisticsParams} from "./params";
import {getCWMetrics} from "./calls";
import {flow} from "fp-ts/function";
import {metricsToStatsResult} from "../transformations/transformations";

const getMetricInfo = (between: Between, fun: string) =>
    (metricName: Metric): TE.TaskEither<string, StatsResult> => {
        const paramsForCW = generateCWMetricStatisticsParams(metricName, between);
        const metricToStatsResult = metricsToStatsResult(fun, metricName);

        return flow(
            paramsForCW,
            p => TE.tryCatch(() => getCWMetrics(p), String),
            TE.map(metricToStatsResult)
        )(fun);
};

export const retrieveMetrics = (between: Between, metrics: readonly Metric[]) =>
    (fun: string): TE.TaskEither<string, StatsResult[]> => {
        return A.array.sequence(TE.taskEither)
            (metrics.map(getMetricInfo(between, fun)));
    };
```

Ok, wow. The imports are probably easy enough, but the rest will definitely need some explaining. We will start with the basics. What is all this TaskEither stuff?

Monads for asynchronous behavior

TaskEither is a Monad provided by fp-ts. Actually, it is a Monad Transformer. We already mentioned how you can actually 'stack' Monads on top of each other to combine their effects. So that helps

you understand why Either is in there. But Task - we haven't yet seen that one, have we?

Task is a Monad for *asynchronous non-failing calls*. A Promise is easily converted into a Task. Just create a function that returns a Promise. That's it.

```
const thisIsATask: T.Task<number> = () => Promise.resolve(42);
```

Why is this Monad useful? *Because it is a Monad*. Meaning that you can do all kinds of transformations, like you would with any other Monad:

```
// result is still T.Task<number>
const result = pipe(
    thisIsATask,
    T.map(n => n + 1) // after getting back the result, we increase it by 1
);
```

Similarly, you can use `chain` to combine multiple Tasks. Which means we now have the power of Monads, their composability and ease of use for o-so-common asynchronous Node functionality. Eventually, when you actually need the results, call the Task as a function and it will return the Promise it contains.

```
// now result is Promise<number>
const result = pipe(
    asTask,
    T.map(n => n + 1)
)();
```

Before we continue, a small sidetour. As we mentioned several times before, Functional Programming tries to keep everything pure and thus easy to reason about. Side effects and I/O ruin this because they add impurity to your program. Your call will give different results, despite having the same input, if it uses I/O (for instance, if the Lambda we are monitoring gets called more, we will get back a different sum). Which is why FP does the best it can and moves all this stuff 'to the edge'. So despite what you might be expecting, this:

```
const thisIsATask: T.Task<number> = () => Promise.resolve(42);

const result = pipe(
    thisIsATask,
    T.map(n => n + 1)
);
```

is pure because nothing is actually happening. There is no asynchronous behavior (i.e. a Promise) being run. No, we are creating a Task and telling our code to perform a map on it when it is run. *But it does not start running yet*. You can test this by adding a `console.log` statement. Nothing will be printed.

```
const asTask: T.Task<number> = () => {
    return new Promise((resolve) => {
        console.log('Doing stuff')
        return resolve(42);
    });
};

const result = pipe(
    thisIsATask,
    T.map(n => n + 1)
);
```

At some point, you will actually have to run the Task though - else, why did you create it? But FP leaves this responsibility to the user of the application. So you actually want to get things done? Fine, just invoke the function and let it be on your head!

```
const result = pipe(
    thisIsATask,
    T.map(n => n + 1)
)();
```

And you will see the logging appearing. The consequence of this is that the program we are writing in this chapter will be pure, free of side effects. That is, pure until, at the very last step, the function is invoked to get back a Promise, containing the results we so desperately desire. Or another way to put it: we are, at a high level, describing what we want the code to do. When it is run, an 'interpreter' will decide what actually has to happen. And it will decide whether it needs side effects to accomplish this. If the interpreter is a mock one, it might not need any side effects! But real interpreters probably will.

Right, back to the subject at hand. So we know what a Task is, we already knew Either. What is their combined power? Easy: handling asynchronous tasks *that might fail*. Instead of writing a `try/catch`, you use the TaskEither Monad and it will either return an error when the catch is triggered, or give you back your result. And our call to AWS can definitely fail, for example if we made a mistake in the name of the function.

Ok, so here is the first function from our file again:

entrypoint.ts

```
const getMetricInfo = (between: Between, fun: string) =>
    (metric: Metric): TE.TaskEither<string, StatsResult> => {

    const paramsForCW = generateCWMetricStatisticsParams(metric, between);
    const metricToStatsResult = metricsToStatsResult(fun, metric);

    return flow(
        paramsForCW,
        p => TE.tryCatch(() => getCWMetrics(p), String),
        TE.map(metricToStatsResult)
    )(fun);
};
```

We can see that the call takes a `Between`, the function name and a metric name. It returns a TaskEither, with the error as a string (the left) and the successful result as a `StatsResult`. What about `paramsForCW` and `metricToStatsResult`? Both are partially applied functions. The parameter generation function has received a metric name and between, but is still missing a function name. `metricToStatsResult` has the function and metric names, but is waiting for the output of the Cloudwatch call. So both these `const` values are functions waiting for just one more parameter. We have used this technique of partially applying before, in our introduction, when we were demonstrating composing.

Next, `flow`, which is a helper from the fp-ts library for composing functions. It is actually just `compose`, but *applied from left to right*. It starts generating parameters, passing those to our cloudwatch function which is inside a TaskEither 'constructor' (lifting function) `tryCatch`. The latter will - eventually - run what you pass in (i.e. our Promise-generating cloudwatch function) and in case of failure it will use the constructor you gave as a second argument to build an error. Here we pass the `String` constructor to get the error as a string. Note that `tryCatch` actually requires a `Lazy<Promise>` as the first parameter. In practice, this just means you pass in a function that returns a Promise. Because that is in effect lazy, i.e. not evaluated immediately.

Programming languages can be sorted by their type of evaluation, eager or lazy, Haskell being one example of the latter. Basically, in a lazy language something is only evaluated *when necessary*. This might be/seem more complex, but it has its advantages. For one, there are potential performance gains since whatever you do not need is never evaluated. (Java developers can compare with Hibernate's eager versus lazy fetch modes.) Another advantage is the ability to work with infinites, for example taking the first five elements of an infinite list. Your program would never come to an end if you had to evaluate the entire list! A lazy language, on the other hand, just evaluates the first five elements - which is all it needs at that point - and returns those.

Next, we use TaskEither's `map` to transform the output into something we can use, the `StatsResult` type we defined earlier. Finally, but importantly, we start the flow by feeding in the function name

`fun`. Without that function call, nothing would happen - we would just have a function waiting for an argument.

Since `flow` is just a compose in a different order, you also know why we partially applied our functions earlier: only one parameter is passed to the next function, but two of our functions required three parameters! So we gave them the two we already have available, knowing the flow would deliver the third and final one.

Note that I originally wrote the above flow like this:

```
// other getMetricInfo code

return flow(
    paramsForCW,
    getCWMetrics,
    p => TE.tryCatch(() => p, String),
    TE.map(metricToStatsResult)
)(fun);
```

And though I like this version because you could further abstract away the `tryCatch` as a 'utility' with a predetermined error constructor, *this is not actually pure* because a Promise is created as soon as our function is called. The previous code, on the other hand, was lazy. And by postponing creation it stayed pure.

Also, you may be wondering about the `map` with `metricToStatsResult`. Is this the right location for that transformation? Isn't the goal of this module just to get information out of Cloudwatch? A valid point. On the other hand, in this way AWS types remain an implementation detail - not visible to whoever uses the 'entrypoint' functions. Though in that case, to be entirely consistent we should also remove Datapoints from the returned type. So getting rid of that is a possible improvement.

Onto the second entrypoint function.

entrypoint.ts

```
export const retrieveMetrics = (between: Between, metrics: readonly Metric[]) =>
    (fun: string): TE.TaskEither<string, StatsResult[]> => {
        return A.array.sequence(TE.taskEither)
            (metrics.map(getMetricInfo(between, fun)));
};
```

First off, note the `export`. This function is exported and is the real entrypoint of our gateway module. Because the code is a bit dense, the body has been split up into two lines. The signature should be obvious by this point. And the parameters are similar to those we gave the other function, except now we are expecting an array of metrics instead of a single one. So all the metrics we desire can now be passed into this function. Similarly, the return has changed into an array - of `StatsResult`.

To get a slightly 'nicer' result, we could have create a `StatsResults` (note the final *s*) as an alias for this array.

What does `A.array.sequence` from the Array module do? The naming might be a giveaway: it will run one or more functions of the type given as first argument, in this case `TaskEither`, *in parallel*. So this is similar to running a `Promise.all` but - mainly thanks to this being a Monad - even easier to use. All of the difficulties of parallelism have been abstracted away.

The second argument is the array of TaskEithers we want to run. We want to run the function for retrieving metrics once for every metric (so three times in total). To avoid some typing we can use partial application and pass in `getMetricInfo` with two out of three arguments already provided. Which means we can use point-free style and do not have to pass in the metric name explicitly inside the `map`. But if you prefer to be explicit, you can replace that last line with `metrics.map(m => getMetricInfo(between, fun)(m))`.

And that is it. Now you are all caught up and we have finished creating our entrypoint for retrieving all relevant metric information from Cloudwatch. And despite this being a more 'real' example than that from our previous chapter, we are still writing in the same style. I.e. using short, understandable functions that combine to form larger functionality. This time they were spread out over more modules/files though.

Moreover we were able to keep a lot of our code pure despite the fact that the core functionality of this application is about retrieving information from external services, which is impure. Our new Task Monad and Monad (Transformer) TaskEither helped us control this potentially dangerous type of code.

Transforming our data

Now we have a TaskEither returning - if everything went well, we are ignoring the error path for now - a `StatsResult` array, with each element of the array containing information for a different type of metric (one will, for example, contain data for errors). So how do we transform our individual `StatsResult` elements to an output useful to us, a developer keeping an eye on his applications? Simple: we will use small, composable, pure functions. And we already have a folder and file called transformers, so why not add it all in there, right?

Ok, so first, we want the total sum of every individual metric. For clarity's sake, we will dissect the AWS `Datapoints` type we are currently returning within `StatsResult`:

cloudwatch.d.ts (in the aws-sdk)

```
export type Datapoints = Datapoint[];

export interface Datapoint {
    Timestamp?: Timestamp;
    Sum?: DatapointValue;
    Average?: DatapointValue;
    // other values
};

export type DatapointValue = number;
```

If a library has types that it wants to expose to its users, these are typically contained inside a file ending with d.ts. Hence the name of the cloudwatch types, 'cloudwatch.d.ts'.

So our `Datapoints` value is an array of `Datapoint` and each Datapoint contains the `Sum` information we want to extract, which is a `DatapointValue`, which is itself an alias for `number`. Note that because we set the `Period` parameter quite high early on, it is possible that there is only one number present within the array - in which case we could avoid a lot of work. But we can't be sure and assuming this might also cause some icky bugs if someone changes either the period or timeframe (we are assuming it to be a few hours, but what if someone changes it to a few days? in that case the probability of an array with multiple elements is very high). So we will assume that there might be more than one `Datapoint` and thus more than one `Sum`.

Now that we know what our type looks like, we add a new type that we want to create out of our current ones:

types.ts

```
export type StatsSumResult = {
    readonly functionName: string,
    readonly metric: Metric,
    readonly sum: number,
}
```

As you can see, it is the same as before but now the sum is just one number. So we should write a function which consolidates `Sum` elements into a single number.

transformations.ts

```
import {fold, monoidSum} from "fp-ts/Monoid";

export const statsResultToStatsSumResult =
    (results: readonly StatsResult[]): StatsSumResult[] => {
        return results.map(b => ({
            functionName: b.functionName,
            metric: b.metric,
            sum: fold(monoidSum)(b.data.map(d => d.Sum)),
        }));
    };
```

A seemingly simple function, since for the most part it just maps our current array to a new one of type StatsSumResult. Except… what the hell is a `fold`? What is `monoidSum`? Is that similar to a monad?

Remember the semigroup from an earlier chapter? Here is a quick reminder: a semigroup provides a way of combining two values of the same type, resulting in a single value of that type. Product, which combines two numbers using multiplication, is an example. **Monoids** are semigroups *with the addition of an empty value*, a starting element. That starting value is 'neutral', which means it does not care what other value you combine it with - it will just give back the other value as a result. So the Monoid for Product would have 1 as its empty value/starter (7 * 1 = 7), while Sum would have 0 (7 + 0 = 7). `monoidSum` is simply a utility function from fp-ts, providing a Monoid sum out of the box.

And why does the functional world like monoids? Well - here we go again - because they are composable. If I create a monoid for combining strings and another for combining numbers, I can use them as components when I want to handle more complex structures that contain those primitives. No additional logic required. We will see an example of this later. As an additional plus, operations on monoids can be made parallel with relative ease. Take sum as an example, with the values 1 up to 4. You can add those numbers in one go, or you could sum 1 + 2 (=3) and 3 + 4 (=7) in a first phase, adding the results of those calculations in a second phase, 3 + 7.

Fold is very similar to reduce, which you might know from Javascript or other languages. It is a function that takes a starting element and a function (so it's another one of those higher-order functions) which knows how to combine the 'accumulated' value with values from a list/array, which are fed in one by one. If we want to get the sum, this function would add the 'current' element of the array to the accumulated value. Fold and reduce are very versatile. You can quite easily create functions like filter or map with reduce. Try it!

Fold and reduce are often used together with an array, which is why I give that as an example. But you can also fold monoids and monads! Also note that there are often a fold left and a fold right, differing only in starting element (left side of the array, or the right) and direction.

And now we know what the above code does. It uses a fold to reduce the given array with monoidSum. Meaning it will start with 0 and combine elements through addition.

Since we now have a type we can work with, let us think about what we want to show users of our application. What we would like to show is how healthy the Lambda is. As mentioned earlier, we will do this with a `status`, which can take three different values: OK, WARN and ERR. This is good for a brief overview of the situation. For more details, we will also add a `message` which can contain further information.

types.ts

```
1  export type Status = 'OK' | 'WARN' | 'ERR';
2
3  export type ReducedStats = {
4      readonly status: Status,
5      readonly message: string,
6  }
```

As usual we limit our `status` to the types we specified because that makes the field easier to work with. Messages are not as easy to limit in value so we will let `message` be a string. The next big piece of the puzzle is a function to transform a single `StatsSumResult` in a `ReducedStats`.

transformations.ts

```
1  const statsSumToStatusAndMessage = (invokes: number) => (s: StatsSumResult) => {
2      let status: Status = 'OK'; (1)
3      let message = ''; (1)
4
5      switch (s.metric) {
6          case 'Invocations': {
7              status = invokes === 0 ? 'WARN' : 'OK'; (2)
8              message = invokes === 0 ? 'not invoked' : 'invokes ok'; (2)
9          }
10             break;
11         case 'Errors': {
12             const percentage = invokes !== 0 ? s.sum / invokes * 100 : 0; (3)
13             status = percentage >= 5 ? 'ERR' : 'OK';
14             message = percentage >= 5 ? 'throwing errors' : 'no errors';
15         }
16             break;
17         case 'Throttles': {
18             const percentage = invokes !== 0 ? s.sum / invokes * 100 : 0; (4)
19             status = percentage >= 1 ? 'WARN' : 'OK'; (4)
20             message = percentage >= 1 ? 'being throttled' : 'no throttles'; (4)
21         }
22             break;
23         default:
```

```
            const _exhaustiveCheck: never = s.metric; (5)
            return _exhaustiveCheck; (5)
    }

    return {
        status, message,
    } (6)
};
```

(1) We start with an empty `message` and an OK `status`, which can be overwritten by the switch that follows. We have to tell Typescript that status has the `Status` type, else it will assume this is a string and downstream code will not accept this string as equivalent to `Status`. Ideally, the compiler would notice that only three possible values, matching those of our type, appear in this piece of code and conclude that this is equivalent to `Status`. But Typescript is not there yet. Also, because `Status` does not have an empty string value, we cannot initialise it as such. So we start with the reasonable default of 'OK'.
(2) We are using a switch to check every possible value for `Metric`, so (if you recall from earlier this chapter) three in total. This is the closest we can get to the power of pattern matching in other functional languages. Notice how, when we get an 'Invocations', we already have the answer ready. Why? Because we need the number of invocations in (3) and (4) as well, so we added it as a first parameter to this function.
(3) Here we handle Error and *report an error when they exceed 5% of the total invocations of the function*. Which is why we take the sum, divide by the invokes, multiply by a hundred, and check the percentage we get. If there are no invocations, there will not be any errors either, so in that case, we return a 0. Note that the five percent threshold is a magic number and we might want to make this configurable. That is quite easy to do via Lambda environment variables, which we will use later in this chapter for getting the function name. If you don't think these values will change often, another approach could be to move the values to the constants file.
(4) Similar to the errors, except here we give back a warning when we have 1% (or more) throttles, but we only give a warning, not an error status. An occasional throttle is nothing unusual.
(5) What is this? It's an exhaustiveness check, which we already encountered once before. Say we had a fourth metric, called MagicInvocations - invocations of your Lambda that come out of nowhere. If MagicInvocations was a possible Metric type value, this default would try to assign it to `_exhaustiveCheck`, which has type `never`. And you cannot assign something to never. So this check will throw a warning at compile time if you add a metric but forget to handle it. Nice. Do note that an IDE might not understand this and might advise you to remove or change this default case in a way that will stop it from working.
(6) We passed through the switch, so `status` and `message` now have the correct values, which we can return.

You may have noticed we are mutating two variables within this function! Isn't that a horrible mistake? While a different approach which does not mutate variables might turn out cleaner, the important thing to note is that we are only mutating *local* variables. The function creates and

manipulates them, no one else. Which means we will not run into any issues with this mutability. If the golden rule is to avoid mutation, this is the addendum: if you do mutate variables, limit the scope of that mutation. Global variables, as you know, are extremely bad. Unexposed mutation within a function is acceptable.

Ok, so now we can produce a bunch of messages and strings. But we actually just want to output ONE status and ONE message. How do we combine them? Well, how about using `fold` and a monoid? How would we go about combining two statuses? Well, clearly if B says everything is ok and B says there is a warning, we want there to be a warning! Similarly, we prefer an error to a warning. So we get the following:

transformations.ts

```
import {fold, monoidSum} from "fp-ts/Monoid";

const statusMonoid: Monoid<Status> = {
    concat: (x, y) =>
        x === 'ERR' || y === 'ERR' ? 'ERR'
            : (x === 'WARN' || y === 'WARN' ? 'WARN' : 'OK'),
    empty: 'OK'
};
```

The monoid has a `concat` and an `empty`. Remember how the empty value is always overwritten by any other value? So here the empty is OK, since we always prefer a warning or error to that value. The `concat` is written somewhat densely, but it just shows what we just said: error > warning > ok.

Now a monoid for the messages:

transformations.ts

```
const messageMonoid: Monoid<string> = {
    concat: (x, y) => (x + ' ' + y).trim(),
    empty: '',
};
```

Not perfect, but good enough. The empty string will be overwritten by other values and we can combine two strings by appending. As we said earlier, Monoids have the nice quality that they are composable. So how do we go about doing that? Fp-ts to the rescue!

transformations.ts

```
import { getStructMonoid } from 'fp-ts/lib/Monoid'

const ReducedStatsMonoid: Monoid<ReducedStats> = getStructMonoid({
    status: statusMonoid,
    message: messageMonoid
});
```

Nice. Using `getStructMonoid`, we say that the monoid for `ReducedStats` uses the above two 'primitives' for its fields. The rest happens automagically. Now, to bring everything together.

transformations.ts

```
export const statsReducer = (results: readonly StatsSumResult[]): ReducedStats => {
    const invocations = results
        .filter(m => m.metric === 'Invocations')
        .map(m => m.sum)
        .shift() || 0; (1)

    const singleStatsSumMapper = statsSumToStatusAndMessage(invocations); (1)

    return fold(ReducedStatsMonoid)
        (results.map(singleStatsSumMapper)); (2)
};
```

(1) We saw earlier that our `statsSumToStatusAndMessage` needs the number of invocations, so we retrieve those. If the value is not present in our array, `shift` will return undefined and we return `0` instead. We could also have used the `??` operator from Typescript, but in this case both are equally valid.
(2) Next, we give the monoid we just wrote to `fold` and pass in all our `StatsSumResult` mapped to `ReducedStats`. `fold` will reduce those to a single status and message. It's another nice example of the way FP allows us to write code at a higher level of abstraction. We just say we want the `ReducedStats` to be reduced into a single value. How this is done is none of our concern.

There are other equally valid ways to reduce the stats to a status plus message. Like a forEach, map or… reduce. But this was a nice opportunity to show Monoids in action.

One thing left to do with regards to transforming: we want to give back results to the outside world, so just in case someone puts us behind an AWS API Gateway - and uses REST calls - we will add a status code and related fields.

types.ts

```
type StatusCode = 200; (1)

export type HttpResult = {
    readonly statusCode: StatusCode,
    readonly body: string,
    readonly headers: Record<string, string>,
}; (2)
```

(1) For now only one relevant code, for when everything is fine.
(2) We will give back a http status code, the results in the body and headers (those might be needed for stuff like CORS).

To go from our reduced stats to this requires only a little work.

transformations.ts

```
export const httpResultFromStats = (result: ReducedStats): HttpResult => ({
    statusCode: 200,
    body: JSON.stringify(result),
    headers: {
        'Access-Control-Allow-Origin': '*'
    }
});
```

Note that when you return this from a Lambda, AWS will know how to interpret these values. It will see `statusCode` as a http status code. Similarly, everything inside `headers` will be added as headers to the response. In case we decide to talk with browsers, we are allowing every origin to call us. Nice and secure… With that we've come to the end of our happy flow. But apart from error handling, which will turn out to be quite easy, a few pieces are still missing. Going way back to our Cloudwatch entrypoint, we see that we need a `Between` type and a function name. Where do those two come from?

Synchronous behavior and IO

We will ignore the function name for now and focus on how to create the `Between`. As you may have guessed, there are problems with purity when it comes to getting the start and end times because we want to show a certain range *with regards to the current time* - for example the last three hours of metrics. This means that the function for getting our `Between` will return different values depending on when it was invoked - which is impure behavior. And so we will wrap it in… what? A Task? Well, we could do that, but this is synchronous behavior, for which the **IO** Monad is ideally suited. Similarly, IO is also what you normally use to wrap your logging, which is after all another synchronous side effect.

As others have noted, in a hybrid language like Typescript (which allows for others styles of programming besides FP) we can decide for ourselves which effects are important enough that we want to keep an eye on them. And that decision will/should come down to the following: does the correct behavior of our program depend on the effect, in this case logging? In most cases it won't, but what if our console output is used by another program as input? In the latter case, our output is an essential part of our application and should be treated with the utmost care. Thus, if you like debugging with console statements and all that logging is temporary, go ahead and take advantage of the flexibility of Javascript which does not force you to wrap logging in a Monad. If you are capturing some important logs but your application does not depend on them that seems - heresy - acceptable to me as well. On the other hand, in some rare cases (like the first one we discussed) not using IO might prove to be a mistake.

As with several other values that could potentially be configurable, we will keep it simple and just check for the last three hours. So we add these values to our constants.

constants.ts

```
1 const ONE_HOUR_EPOCH_MILLI = 60 * 60 * 1000;
2 export const THREE_HOURS_EPOCH_MILLI = 3 * ONE_HOUR_EPOCH_MILLI;
```

You may already be wondering: if we were to make a lot of these values (error percentage, function name, times) configurable, how would we go about passing them to so many functions? Won't that become very complicated? Part of the answer will be to gather them in a `Config` type, meaning we only need to pass in one clearly defined value. And if they are needed by a lot of functions, we would use the **Reader** Monad. Which we will be discussing later. Stay tuned!

Now we create our function to get a `Between` wrapped in an IO. First, create a new file called time.ts in `util` (i.e. the same folder the constants are located in).

time.ts

```
1 import {Between} from "./types";
2 import {THREE_HOURS_EPOCH_MILLI} from "./constants";
3
4 export const dateToBetween = (curr: Date): Between => ({
5     startTime: new Date(Number(curr) - THREE_HOURS_EPOCH_MILLI),
6     endTime: curr,
7 }); (1)
```

(1) Creating a `Between` *based on an incoming value* is pure. Given a certain date (say the first of January 2020), it will return the same `Between` every time! So we extract pure from the impure and put the latter in a separate function.

We want to pass on both function name and this between value. So we add a type for configuration.

types.ts

```
export type Config = {
    between: Between,
    functionName: string,
};
```

To keep things simple `functionName` is just a string. Now to create a config.ts file, again in the `util` folder:

config.ts

```
import {io, IO, map} from "fp-ts/IO";
import {Between, Config} from "./types";
import {create} from "fp-ts/Date";
import {dateToBetween} from "./time";
import {pipe} from "fp-ts/pipeable";
import {sequenceT} from "fp-ts/Apply";

const addToConfig = (between: Between, functionName: string) => ({
    between,
    functionName,
}); (1)

export const generateConfig = (): IO<Config> => {
    return pipe(
        sequenceT(io)(
            map(dateToBetween)(create), (2)
            () => process.env.FUNCTION_NAME,
        ), (3)
        map(([a, b]) => addToConfig(a, b)), (4)
    );
};
```

(1) A pure function for creating our config.
(2) A helper from the fp-ts library, `create` *"Returns the current Date"* wrapped in an IO Monad.
(3) Which means we have to lift it into the config. Similarly, we get the function name from our environment (more on that later), so we create an IO for that too. We could use `IO.of()` to do that, but since IO is simply a function returning a value, we do that instead.
(4) Next we use `map` to retrieve the two values and pass them to our pure `addToConfig`.

So we used two small pure functions and some glue from fp-ts to create our functionality. Again, it's turtles… eh… composition all the way down.

Bringing it all together

Now, let us try to bring our entire program together for the first time. We want our handler to look like this:

```
// some imports

export const handler = async () => {
    return pipe(
        generateConfig(),
        T.fromIO,
        T.chain(checkLambda),
        T.map(httpResultFromStats),
    )();
};
```

To make inference and usage of fp-ts simple we again resort to using `pipe`. Inside that pipe, we generate the configuration and get back an IO. But our other actions are all asynchronous and therefore Tasks! So how do we deal with this mismatch? One way to do it is to 'lift' our IO stuff to Task. For that we can use one of the built-in functions from Task, `fromIO`. From that point on, we are working with a Task, which is why we use its `chain` to pass our config to an as-yet-unwritten `checkLambda` function.

We could also have used Tagless Final to abstract away the concrete Monads we are using, but that is a more advanced technique that we will briefly discuss later on.

This function will run our metric retrieval and transformations for the Lambda which is passed in via config. It will therefore take in a config and return `ReducedStats`, which will then be mapped to an http result using a function we wrote earlier. So let us fill in the implementation:

```
const checkLambda = (config: Config) => {
    return pipe(
        retrieveMetrics(config.between, METRICS)(config.functionName),
        TE.map(statsResultToStatsSumResult),
        TE.map(statsReducer),
        TE.fold(
            (message) => T.task.of(reducedStatsForFailure(message)),
            T.task.of),
    );
}
```

We are combining the functions we wrote in the previous pages. We pass our metrics, time and function name to our `retrieveMetrics` call. Next we transform those results twice, using `map` and two of the pure transformers we defined earlier. One minor issue, similar to before: because our call could fail, we are dealing with a TaskEither! How do we go from this monad to a Task? Here we chose to use `fold` which in the case of a TaskEither requires two functions, one for handling the left value and one for the right. Right is easy, just put it inside another Task! On the left side, though, we only have a string containing an error message. So here we take that message and pass it into a new function called `reducedStatsForFailure`, which is placed in the transformers folder and looks like this:

transformations.ts

```
1  export const reducedStatsForFailure = (message: string): ReducedStats => ({
2      status: 'ERR',
3      message: `Could not retrieve info for this function. Error: ${message}`,
4  }); (1)
```

(1) If we encounter an error, we definitely want to report it, so we set the status to ERR and add a message.

So in both cases we get back a Task containing `ReducedStats`, but in one case it contains a message with information about the function, in the other case we have an error because we could not retrieve information from AWS.

Put together this gives us the following index (which you can place directly under the `src` folder).

index.ts

```
1  import * as TE from 'fp-ts/lib/TaskEither';
2  import * as T from 'fp-ts/lib/Task';
3  import {Config} from "./util/types";
4  import {METRICS} from "./util/constants";
5  import {retrieveMetrics} from "./metric-gateway/entrypoint";
6  import {pipe} from "fp-ts/pipeable";
7  import {
8      httpResultFromStats,
9      reducedStatsForFailure,
10     statsReducer,
11     statsResultToStatsSumResult,
12 } from "./transformations/transformations";
13 import {generateConfig} from "./util/config";
14
15 // the function we just discussed
16 const checkLambda = (config: Config) => {
17     return pipe(
```

```
        retrieveMetrics(config.between, METRICS)(config.functionName),
        TE.map(statsResultToStatsSumResult),
        TE.map(statsReducer),
        TE.fold(
            (message) => T.task.of(reducedStatsForFailure(message)),
            T.task.of),
    );
}

// our handler, discussed earlier
export const handler = async () => {
    return pipe(
        generateConfig(),
        T.fromIO,
        T.chain(checkLambda),
        T.map(httpResultFromStats),
    )(); (1)
};
```

(1) Note the additional `()`. To actually get a result, we need to go back to ordinary Javascript. And since a Task is simply a function which returns a Promise, we just have to call the result of our piping to get that Promise. Which we return from the handler.

So our index does the high-level glueing. It generates config, retrieves the metrics for our Lambda and returns it as an HTTP result. If a failure occurs, we output an HTTP result with an error message. Finally, we return a Promise, which AWS Lambda knows how to handle.

The only thing left to do, is to add the function name as an environment variable. Which is not too hard, it only requires a minor change in our infrastructure:

infrastructure.yaml

```
AWSTemplateFormatVersion: 2010-09-09
Transform: AWS::Serverless-2016-10-31

Resources:
  MetricsLambda:
    Type: AWS::Serverless::Function
    Properties:
      Handler: main.handler
      Runtime: nodejs12.x
      MemorySize: 512
      Timeout: 20
      CodeUri: dist/cloudwatch.zip
```

```
13      Policies:
14        - AWSLambdaExecute
15        - Version: "2012-10-17"
16          Statement:
17            - Effect: Allow
18              Action:
19                - "cloudwatch:Get*"
20                - "cloudwatch:List*"
21              Resource:
22                - "*"
23      Environment: (1)
24        Variables:
25          FUNCTION_NAME: monitoring-example-MetricsLambda-123456789
```

(1) New stuff starts here. We tell AWS Lambda that we would like to add an environment variable called `FUNCTION_NAME`, with the given value, to our Lambda environment. (Of course the value of `FUNCTION_NAME` will be different in your case. Check your CloudFormation or Lambda console.)

Ideally, you would not refer to a hardcoded value in your yaml file. You could write code to retrieve the correct name. Or if the Lambda you want to check is in the same template as your MetricsLambda, you could use Ref[14] to get the right name injected by CloudFormation. In this case though, because we have Lambda which is observing itself, that would give a circular dependency and would fail. But in real scenarios, that would be the way to go.

That's it. You can now deploy the Lambda again and check out what results it gives when you trigger it.

A few remarks before we continue. First, there were several possible and valid ways to use the functions defined in the transformations folder. Here we chose to combine them at the very last moment. We could have used compose/flow to reduce the number of maps needed in our index. Like so:

transformations.ts

```
1 export const combined = (results: readonly StatsResult[]): ReducedStats => {
2     return flow(
3         statsResultToStatsSumResult,
4         statsReducer,
5     )(results);
6 };
```

Which is one less map in checkLambda. Or we could have gone a step further still:

transformations.ts

[14] https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/intrinsic-function-reference-ref.html

```
export const combined = (results: readonly StatsResult[]): HttpResult => {
    return flow(
        statsResultToStatsSumResult,
        statsReducer,
        httpResultFromStats,
    )(results);
};
```

In which case we would remove the last map in index.ts and transform the failure to `HttpResult` in `checkLambda`. Both approaches (i.e. composing with maps or with flow) have their advantages and disadvantages. Composing within the transformers file has the advantages in exposing less surface, thus resulting in a better defined module. I.e. transformers is a module that takes care of changing your result in something you can give back - and you have no business knowing how exactly it goes about doing this. That's an implementation detail.

On the other hand, that approach can be too rigid. What if this code was shared by two Lambdas and the second one wanted to use `ReducedStats` to generate an email - meaning it has no need of the final `HttpResult` transformation? We would have to add another function to transformers, composing the first two and adding a new third function. Or, if we had exposed more functions, we could leave it up to the Lambdas themselves to use the transforms available to get the right results. So minimizing the surface area (the number of exposed functions) is a good idea, but you still want to have a bit of flexibility. So it depends, but obviously in the end you will be combining functions, one way or the other.

It is also a best practice not to put too much logic in your index file when writing Lambda code. Why? Because it often contains a lot of initialisation stuff, which might make whatever is in that file difficult to unit test. So we could extract `checkLambda` into another file. Also, note that a Lambda 'container' can be reused by subsequent invocations. If I have a request A at time `0` and a request B at time `1`, any logic outside of the `handler` function will be initialised *only once*. Which means a Lambda with the following code, when invoked twice…

```
console.log(`current time is ${Date.now()}`);
export const handler = async () => {
    return pipe(
        // ...
    )();
};
```

… *might* print the same time multiple times, depending on whether you received a new container from AWS, or the old one. This means you can optimise your code by doing as much initialization as possible outside of your handler, because anything outside the handler is subject to reuse. In our case, though, there is little to optimise. We could create the config beforehand, but we would risk getting old `Between` values again and again. Because in a really bad/good - heavily dependent on your perspective here - case, a Lambda container could be reused for hours.

Finally, a nice minor improvement would be to make a newtype `FunctionName` to replace the string from `Config`. That way you make sure no-one passes some faulty string to the relevant functions by mistake.

Extensions

We can now monitor one function. What kind of functionality are we likely to add in the future? One very plausible extension would be monitoring multiple Lambdas. How would we go about doing this? Well, it would require multiple minor modifications, with our type system helping us discover what to change.

First, since we now have multiple functions to monitor, we would change our configuration to contain an array of strings (or newtypes), causing compile issues in `Config`. For now, we could solve these by changing `addToConfig`.

types.ts

```
export type Config = {
    between: Between,
    functionNames: string[], // also note that the property is now plural!
};
```

config.ts

```
const addToConfig = (between: Between, functionName: string) => ({
    between,
    functionNames: [functionName],
});
```

Right now there is *still* only one function to track, but if we change our code to work with an array, adding more functions is easy. The easiest way to get multiple names at this stage might be to change our environment variable to a comma separated list - splitting on the comma in our code. We will be ignoring these issues for now because our compiler has started complaining about `retrieveMetrics` (in `index.ts`). So change `config.functionName` to `config.functionNames` and make `retrieveMetrics` accept an array. Now some further changes are required. (I shortened the parameter names because the signatures are quite long.)

entrypoint.ts

```
const retrieveMetricsForFunction = (b: Between, m: readonly Metric[]) =>
    (fun: string): TE.TaskEither<string, StatsResult[]> => {
        return A.array.sequence(TE.taskEither)
            (m.map(getMetricInfo(b, fun)));
};

export const retrieveMetrics = (b: Between, m: readonly Metric[]) =>
    (fun: string[]): TE.TaskEither<string, StatsResult[][]> => {
        const genericRetrieveMetrics = retrieveMetricsForFunction(b, m);

        return A.array.sequence(TE.taskEither)
            (fun.map(genericRetrieveMetrics));
};
```

What we did here was add `retrieveMetricsForFunction`, which is identical to the original `retrieveMetrics`. The modified `retrieveMetrics` now first partially applies the arguments that are the same for each function (the dates and the metrics to retrieve) to `retrieveMetricsForFunction`. Next we `map` over our function names and pass them to the new helper. All of this is again done in parallel using `sequence`. Next, the return type started complaining. And rightly so, because we now have a *nested* array for StatsResult. So we fixed that as well.

Now we go back to our `checkLambda` which is complaining because it is now receiving a nested array... We only need a small change to get it compiling again.

index.ts

```
import {flatten} from "fp-ts/Array";
// other imports

const checkLambda = (config: Config) => {
    return pipe(
        retrieveMetrics(config.between, METRICS)(config.functionNames),
        TE.map(flatten), // this was added
        TE.map(statsResultToStatsSumResult),
        TE.map(statsReducer),
        TE.fold(
            (message) => T.task.of(reducedStatsForFailure(message)),
            T.task.of),
    );
}
```

We `flatten` (you could also have used `flat` from Javascript) the array and now everything works again! So by following the trail of the Typescript compiler, we got things basically in working order again. I say basically, because you might not like the messages we are producing. For example, you

will get reports on applications with errors without knowing *which* Lambda is having issues! This is because we are reducing all incoming metrics without any regard for which Lambda they are coming from.

The work required for making the messages clearer is less than you might think. Remember, we were already passing in the function name in our various types, so once we get here...

```
const statsSumToStatusAndMessage = (invokes: number) => (s: StatsSumResult) => {
    // ...

    switch (s.metric) {
        case 'Invocations': {
            status = invokes === 0 ? 'WARN' : 'OK';
            message = invokes === 0 ? 'not invoked' : 'invokes ok';
        }
            break;
        case 'Errors': {
            const percentage = invokes !== 0 ? s.sum / invokes * 100 : 0;
            status = percentage >= 5 ? 'ERR' : 'OK';
            message = percentage >= 5 ? 'throwing errors' : 'no errors';
        }
            break;
        case 'Throttles': {
            const percentage = invokes !== 0 ? s.sum / invokes * 100 : 0;
            status = percentage >= 1 ? 'WARN' : 'OK';
            message = percentage >= 1 ? 'being throttled' : 'no throttles';
        }
            break;
        default:
            const _exhaustiveCheck: never = s.metric;
            return _exhaustiveCheck;
    }

    // ...
};
```

...our incoming `StatsSumResult` has the function name and we could easily add it to the messages we produce. This will still only give one global status though, not one per function. If you wanted something like that, you would have to handle metrics per function and add the name of the function to `ReducedStats` - in other words, you would *reduce over the metrics* not over metrics & functions. The http function would remain largely unchanged, because it only needs something to put in its `body`.

So in order to write a clean solution for this feature, one that fits in nicely with our existing code,

we changed three functions (`addToConfig`, `retrieveMetrics` and `checkLambda`) and a type. And only in the case of `retrieveMetrics` were the changes somewhat substantial: about four lines of code. Similarly, adding the function name to the messages we output only required adding it to the messages in one function (`statsSumToStatusAndMessage`). Outputting a status per function - only briefly discussed here - would require a bit more work, but not much.

Of course, our work in `statsSumToStatusAndMessage` was limited because we were already keeping the function name handy! Which might be considered bad practice because *you ain't gonna need it* (YAGNI). Regardless, adding the function name later on would not have been that hard - just follow the same process of adding to the type and fixing the compiler errors.

Now to explore another idea. How about sending an email every hour to concerned developers instead of manually calling the Lambda? How would we go about doing that?

Note that we are ignoring the changes we just made and using our original project again for this new idea.

For actually sending the email, we will be using a managed service of the AWS Cloud, Simple Email Service, also known as SES. We will need to add permissions for our Lambda to use this managed service, as well as adding a schedule for our Lambda. Add the following to your infrastructure:

infrastructure.yaml

```yaml
AWSTemplateFormatVersion: 2010-09-09
Transform: AWS::Serverless-2016-10-31

Resources:
  MetricsLambda:
    Type: AWS::Serverless::Function
    Properties:
      Handler: main.handler
      Runtime: nodejs12.x
      MemorySize: 512
      Timeout: 20
      CodeUri: dist/cloudwatch.zip
      Policies:
        - AWSLambdaExecute
        - Version: "2012-10-17"
          Statement:
            - Effect: Allow
              Action:
```

```
19                - "cloudwatch:Get*"
20                - "cloudwatch:List*"
21                - "ses:SendEmail" (1)
22              Resource:
23                - "*"
24      Environment:
25        Variables:
26          FUNCTION_NAME: monitoring-example-MetricsLambda-1BZCIF924BV37
27      Events: (2)
28        CronEvent:
29          Type: Schedule
30          Properties:
31            Schedule: rate(1 hour)
32            Name: metric-lambda-schedule
33            Enabled: True
```

(1) As we already briefly discussed, a Lambda needs permission for everything it does within your cloud environment, so we need to give it the right IAM policies. This one is new and gives us the ability to send an email from SES. Forgetting to call these permissions is a common mistake, but the error you will see in your logs should point you in the right direction.
(2) This code is also new. It tells AWS that our Lambda will listen to incoming events of the `Schedule` type. And we define that Schedule as something that will run every hour (`rate(1 hour)`). When we deploy this, AWS SAM will create a CloudWatch Event that will call our Lambda for us and the Lambda will run every hour.

That was easy. Now the code. Let us think this through. What do we need? Well, we need to add a call to SES and generate the right parameters! So we start there, but first rename `metric-gateway` to `gateway`. This folder is now the place for *all* calls to external systems, including email functionality.

calls

```
1  import CloudWatch,
2  {
3      GetMetricStatisticsInput,
4      GetMetricStatisticsOutput
5  } from "aws-sdk/clients/cloudwatch";
6  // new imports:
7  import SES, {SendEmailRequest, SendEmailResponse} from "aws-sdk/clients/ses";
8
9  const cloudwatch = new CloudWatch({ region: 'eu-west-1'});
10 const ses = new SES({region: 'eu-west-1'}); (1)
11
12 export const getCWMetrics = (params: GetMetricStatisticsInput): Promise<GetMetricSta\
13 tisticsOutput> => {
```

```
    return cloudwatch.getMetricStatistics(params).promise()
};

export const sendEmail = (params: SendEmailRequest): Promise<SendEmailResponse> => {
    return ses.sendEmail(params).promise();
} (2)
```

(1) We create our SES client, similar to how we created one for Cloudwatch.
(2) We call SES with our parameters and return a promise. This is very similar to what we did for getCWMetrics.

Next, we need a pure function for creating the right parameters:

```
export const generateEmailParams = (message: string) => {
    return {
        Destination: {
            ToAddresses: [
                "xxx@xxx.com", // replace with a real email
            ]
        },
        Message: {
            Body: {
                Text: {
                    Charset: "UTF-8",
                    Data: message
                }
            },
            Subject: {
                Charset: "UTF-8",
                Data: "Status email"
            }
        },
        Source: "xxx@xxx.com", // replace with a real email
    };
};
```

Most of this is boilerplate stuff. We are keeping our example simple by hardcoding the sender and receiver email. This too could be added to the Config type. We are passing in a `message` with type string as the body of the mail. Note that we did not pass in one of the types we defined, like `ReducedStats`. Why? Well, from a domain perspective, sending email is a service located at the edge of our application. So we want to keep our domain related stuff out of it. The email service does not need to know anything about our domain. It just wants a simple string and does not care where it came from. Furthermore, we would be limiting the usefulness of this method if we made the type too specific. So a string seems like a good choice.

If you have not used SES before, you are in 'sandbox' mode and will need to verify your ownership of the email addresses you are sending to and from. Else the code will fail. You can verify an address in the AWS SES console.

Since we only expose methods in this folder from entrypoint, to hide implementation details from the rest of our code, we add a function there:

entrypoint.ts

```
export const sendStatusEmail = (message: string): T.Task<SendEmailResponse> => {
    return () => flow(
        generateEmailParams,
        sendEmail,
    )(message);
};
```

Again, we are keeping things simple by just using a Task. If you configured your Lambda to have the right to send emails, the chance of `sendEmail` throwing errors is small. But if you want to be sure, use a TaskEither. All the other stuff is self-explanatory, *nihil nove*.

Note that we did not mention SES in the name of the exposed function, because the other parts of our code shouldn't care about the kind of external service we are using. And by also hiding the creation of SES parameters, SES client, etc. inside the `gateway` folder, we are also making switching out the email provider for another one easier. Only code within this one folder has to change. We are using information hiding with modules to our advantage. (If we wanted to completely hide SES, we there is at least one minor change still waiting though: change the response type.)

Okay, what are we missing besides the glue? Well, something to change our type to a string! So in our transformers we add the following.

transformers.ts

```
export const reducedStatsToEmailMessage = (result: ReducedStats) => {
    return `Status: ${result.status}. Message: ${result.message}`;
}
```

A nice and simple message. Now we can start gluing:

index.ts

```
export const handler = async () => {
    return pipe(
        generateConfig(),
        T.fromIO,
        T.chain(checkLambda),
        T.map(reducedStatsToEmailMessage), (1)
        T.chain(sendStatusEmail), (2)
    )();
};
```

The only thing that changed here, is that we are now (1) mapping to our new `reducedStatsToEmailMessage` instead of the http result and (2) sending the email. Similar to the previous chapter, we were able to achieve our change with minimal impact on existing functionality. Only in the handler, the final glue of our program, did we need to change one call and add another. Everything else was four small new functions, half of them pure. As we saw before adding functionality to a functional project can be a breeze once you get into the right mindset.

Maybe you think that I am cheating. This is a book after all, so I could have worked out the entire application beforehand. Which as a criticism is only partially true. The previous chapter, for example, was coded on the fly. For this chapter, I wrote the email 'extension' simultaneous with these pages. And the only issue I ran into was the Lambda permissions for email sending, which is why I added a warning about forgetting them. But the functional Typescript stuff just worked. Naturally, this does not mean that every refactor/change will be easy. Just that it *can* be a pleasant experience.

A small aside: memoization

In an earlier chapter, we already briefly discussed how FP can be less performant than other styles of programming in *some* situations. But besides the advantages of high-level programming when it comes to optimisations that an engine might do without us even knowing, there is also one specific and powerful tool available to the functional programmer: **memoization**.

Memoization gains its power from pure functions. Because we know that a pure function's result is only dependent on what we pass in, we can deduce that if we calculate the result for a given set of parameters, we can return that result *whenever the same parameters are passed to us in the future*. So in essence, memoization is a function cache that saves results and returns them whenever a future call uses parameters we've seen before. For compute intensive functions, this can be a huge win, even if it comes at the cost of additional storage. Calculating fibonacci numbers is a typical example for memoization. The same parameters will be passed in more than once, calculating large numbers is compute intensive and the cost of storing them is negligible.

So in our example application, we have two things to consider. One, do we have functions that are likely to be called with the same parameters? Two, are those functions compute intensive?

Somewhat regrettably, none of our functions comply with both requirements. But let us pretend that our `reducedStatsForFailure` function has some important calculations to perform, and memoize it. First, add the helper library fp-ts-std[15] to our dependencies. Then add the following to our transformations:

transformations.ts

```
import { memoize } from 'fp-ts-std/Function'
import { eqString } from 'fp-ts/Eq'
// other imports and code

const reducedStatsForFailureNoMem = (message: string): ReducedStats => {
    console.log('Called no mem'); // added as an illustration
    return {
        status: 'ERR',
        message: `Could not retrieve info for this function. Error: ${message}`,
    }
};

// export the memoized version
export const reducedStatsForFailure = memoize(eqString)(reducedStatsForFailureNoMem)
```

As you can see, memoizing is fairly simple. First, the most complicated part: you pass in an `Eq`. If our `memoize` wants to know when to return a cached result, it has to know when the passed-in parameters are equal to one another! In our case, we are dealing with a string, so we can use a built-in `Eq`, `eqString`, from fp-ts. If you are dealing with more complex structures, that's no problem: we can build these Equals from basic building blocks, very similar to how we created a monoid from smaller monoids. Except this time we call `getStructEq` instead of `getStructMonoid`.

The only other thing we do is pass in the original (renamed) method and export our new and enhanced function. Outside users of the code will not notice a difference, but try calling the method more than once with the same string. You will notice only on the first invocations a log line is produced!

Key points

Here are some of the most important things we learned in this chapter and the previous one:

- write small and preferably pure functions
- compose them using `flow`, `pipe`, monads, etc.
- return functions from functions (currying and partial application), which will simplify usage (no need for one long parameter list) and help with composability

[15] https://github.com/samhh/fp-ts-std

- use monads for handling errors, side effects, asynchronous behavior
- where possible, try to limit the 'surface' of your modules to improve use and maintainability

Summary

In this chapter, we created and deployed a project, written in a functional style. We learned about some new, useful monads and saw how to combine them with things we already knew (composing, mapping, chaining, lifting) to create an actually useful application. And we looked at the ways we could expand its functionality by adding new functions, leaving most of the existing code untouched and using the type system for guidance. And there is still more to explore. Read on.

Testing

Yes, I know. All this talk of real applications - even deploying them to the cloud - and we haven't explored testing yet! Even worse, I have claimed on several occasions that functional code is easy to test - but never actually demonstrated this! Fine, have it your way. We will add tests to our monitoring application, discussing the FP approach along the way.

You should have written your test before writing your production code! This is the TDD way. Yes, that's true. I have to confess my sins: I am an imperfect student of TDD, often writing tests after my production code. But don't let my bad example distract you! Writing tests first is definitely a good idea.

The types of testing

There are several types of testing in programming, each on a different level of complexity and size. **Unit tests** stand on one side of the spectrum. They check one isolated piece of code, a class or a function for example. They are very fast and almost always automated. **End-to-end tests**, on the other side of the spectrum, run against entire applications. They are slow and more often manual. Between these two levels, you have **component and integration tests** which are a level up from the isolated unit tests but will still only verify part of the system, mocking some applications or removing them from the equation. **Smoke tests** might be seen as another category. They work with a real system and their goal is to give quick feedback, alerting you if even basic functionality is not working anymore, i.e. there is smoke coming from the system.

A standard piece of advice is to think of testing as a pyramid, with a large base consisting of lots of unit tests (hundreds, sometimes thousands), fewer component/integration tests and a couple end-to-end tests for the most essential functionality. This is because unit tests are fast, reproducible and relatively easy to maintain, whereas an end-to-end test is often costly and brittle. Changing the id of a single `div` in your frontend might be enough for it to start failing. On the other hand, end-to-end tests do show when integrations between several unit-tested components fail and make sure your users will actually be able to use your application, so they definitely have a lot of business value.

In any case, your best bet is to automate. Most testing work is boring, uncreative and slow if done manually. It is the kind of work best left to a computer, leaving human testers free to focus on the hard-to-automate stuff, or to do more creative, exploratory verification. Unit tests definitely should *never* be done manually. Really only end-to-end tests are candidates for manual work and even here it is better to automate the basic flows.

Since we've been developing applications for use with Lambda, a small note: it is often advised for these projects to focus on unit and end-to-end testing. Why? Because it can be difficult to do integration testing when you have multiple (sometimes hundreds) of Lambdas talking amongst themselves and with managed services. There is a lot you would have to mock away. Instead, why not just deploy your application to the Cloud and run some tests there? This has several advantages: it's more realistic, cheap (pay for what you use - so only when you are actually testing) and serverless apps generally deploy very fast (sometimes taking less than a minute!).

That is not to say that you cannot do integration testing, because there are AWS solutions (like dynamodblocal for mocking DynamoDB, or SAM local) and external ones (like Localstack or Moto). But this requires additional setup and is less realistic than deploying into a test environment, so it is not always worth the effort.

You mentioned TDD?

Right, maybe a brief explanation of that concept is in order. TDD, or **Test Driven Development**, comes from the Extreme Programming movement. Its core idea is simple: create a test that verifies the behavior of the code *before actually writing that code*. Say I want to write a function that will add two numbers. Before writing code for doing that, I will create the following test (using the jest framework):

```
describe('sum tests', () => {
    test('should return two when given one and one', () => {
        const result = sum(1, 1);
        expect(result).toBe(2);
    });
});
```

Obviously, this test will fail, because you do not have a function that adds numbers. You do not even have a `sum` yet. Your first step will be to make your code compile (if you are using Typescript) which you can do by creating a `sum` function that takes two parameters and returns nothing and importing it into this test file. Now your test will run. But it will fail on the `expect` because you are not getting the right result. The next step will be to write just enough code to get your tests to turn green (i.e. succeed). So you could add both numbers using + and return the result. Your test is now green. Huzzah! You finished your first TDD cycle.

For more realistic examples, TDD practitioners will point out that you should write *just enough code* to make your current test work. No more. This means you will develop functions in an iterative fashion, always writing part of the required functionality and then pausing to add another test. You probably guessed TDD likes the *You Ain't Gonna Need It* philosophy. Because of this incremental approach, you will often avoid writing unneeded code.

If desired, you can lift the TDD workflow (write failing test - make it pass - refactor - write another failing test...) to a higher level. Write a failing end-to-end test. Now start writing a unit test and

make it pass. Continue writing unit tests and coding, until you make the end-to-end test pass. Write a new end-to-end and start over.

The nice thing about writing code the TDD way is that once your functionality is written, you already know it works because of the tests! Test coverage will obviously be very high as well. This makes it very easy to refactor because your tests will keep you from breaking existing code. TDD also forces you to think about your functionality before writing it. What kind of components do I need? What will they return when I pass them certain parameters? How will they handle edge cases?

This is, however, also one of the reasons I - and most of the people I have worked with - have a harder time adopting it. We are used to exploring ideas by writing our 'normal' code, which can seem to be easier and faster. The downside is that the resulting code might not be as well thought-out, and tested when compared to that written with TDD. So I do think it's advisable for every developer to give TDD a try. Like FP, it can deliver massive advantages once you do actually 'get it'.

FP and testing

What types of testing do we have in a functional language? Well, the same as discussed, plus a little bit extra. When it comes to end-to-end testing, there is no real difference. At that level, language and paradigm are implementation details. The most obvious differences are at the unit level, where our testing focus is now no longer a class, but a function. More specifically, we focus on testing pure functions.

Let us take a look at our monitoring application. What can we test, what do we want to test? An overview:

- **calls.ts** is not really interesting because you would in effect be testing the AWS SDK. Typescript's compiler offers enough certainty here (mistype the name of `getMetricStatistics` and you will immediately see a warning appearing)
- **params.ts** is important to get right, so yes, we want unit tests. Since it consists of short, pure functions, this should be easy
- **entrypoint.ts** is also useful to test, though it comes with a bit of extra complexity because we have to mock AWS calls
- **transformations.ts**: test it! Pure functions!
- **constants.ts** and **types.ts** are not very interesting. If there are any mistakes, we will notice them elsewhere
- **time.ts** will and can be tested, since it only has a pure function
- **config.ts** can be tested, though the 'current date' stuff does not make this easy
- **index.ts** maybe? As I mentioned, in general index files for Lambda contain stuff that you do not want to unit test because it is too much effort. In our case, it's doable but the added value is limited because we are just combining different functions with the compiler correcting us when we make a mistake

We will once again be using the 'original' version of our application, ignoring our extensions.

So we will be adding tests for the pure functions (in params, transformations, time), plus a few for the impure (entrypoint, config). Before we do anything else though, add the jest and ts-jest dependencies to our package.json, in addition to a simple test command, with a code coverage check enabled.

package.json

```
{
  "name": "leanpub-monitoring",
  // ...
  "scripts": {
    // unchanged, only add the below
    "test": "jest --coverage"
  },
  // ...
  "devDependencies": {
  "scripts": {
    // unchanged, only add the below
    "jest": "^26.0.0",
    "ts-jest": "^26.0.0"
  },
  // ...
}
```

And add a jest config file similar to the following to the root of your project. If you don't do this, you will probably get an error telling you your test files are not plain Javascript. Which is true since they are Typescript files - so we need to transform them to Javascript.

jest.config.js

```
module.exports = {
    roots: [
        '<rootDir>',
    ],
    testMatch: [
        '**/__tests__/**/*.+(ts|tsx|js)',
    ],
    transform: {
        '^.+\\.(ts|tsx)$': 'ts-jest',
    },
};
```

Now we can add a folder called `__tests__` to the root of our project (if you look at the above configuration, you can see our testMatch expects the tests to be located in such a folder). Add a subfolder called `metric-gateway` and place a file called paramsTest in it, with the following content:

paramsTest.ts

```
import {generateCWMetricStatisticsParams} from "../../src/metric-gateway/params";

describe('generate cw metric statistics params', () => {
    test('should generate the right params for invocations', () => {
        const startTime = new Date();
        const endTime = new Date(startTime.getTime() + 60 * 1000);

        const result = generateCWMetricStatisticsParams
            ('Invocations', {startTime, endTime})('aFunction');

        expect(result.StartTime).toEqual(startTime);
        expect(result.EndTime).toEqual(endTime);
        expect(result.MetricName).toBe('Invocations');
        expect(result.Namespace).toBe('AWS/Lambda');
        expect(result.Period).toBeTruthy();
        expect(result.Dimensions[0].Name).toBe('FunctionName');
        expect(result.Dimensions[0].Value).toBe('aFunction');
        expect(result.Statistics[0]).toBe('Sum');
    });
});
```

This generates a parameter call for Invocations and uses jest to check whether all the values we need have been set. This is the classic *Arrange-Act-Assert* setup, separated by whitelines.

How about the other metrics? Do we need to write boring, repetitive code to check those as well? What we need is a *parameterized test*, which you might know from JUnit or other frameworks. In Node, they are really easy to write:

paramsTest.ts

```
import {generateCWMetricStatisticsParams} from "../../src/metric-gateway/params";
import {Metric} from "../../src/util/types";

const metrics: Metric[] = ['Invocations', 'Errors', 'Throttles']; (1)

describe('generate cw metric statistics params', () => {
    metrics.forEach(m => { (2)
        test(`should generate the right params for ${m}`, () => {
            const startTime = new Date();
            const endTime = new Date(startTime.getTime() + 60 * 1000);

            const result = generateCWMetricStatisticsParams
                (m, {startTime, endTime})('aFunction');

            // ...
            expect(result.MetricName).toBe(m);
            // ...
        });
    });
});
```

(1) To make sure everything compiles, we assure Typescript that we have an array of `Metric`.
(2) Next, we just call a `forEach`, generating three tests. If we suddenly want to check four metrics, we only have to change the metrics array.

Now let us add some tests for transformers. Here is the first one:

transformersTest.ts

```
describe('http result from stats', () => {
    test('should return a status code, headers and body', () => {
        const reducedStats: ReducedStats = {
            status: 'WARN',
            message: 'just a message',
        };

        const result = httpResultFromStats(reducedStats)

        expect(result.statusCode).toBe(200);
        expect(result.body)
            .toBe("{\"status\":\"WARN\",\"message\":\"just a message\"}");
        expect(result.headers['Access-Control-Allow-Origin'])
            .toBe('*');
```

```
    });
});
```

Again, Arrange-Act-Assert. The only noteworthy thing is that here too we have to add a type (`ReducedStats`) to what we are passing as a parameter, else Typescript will complain.

Here are the other tests for transformers. Warning, this is a fair bit of code (without anything really interesting, so maybe just skim through it).

transformersTest.ts

```
import {
    httpResultFromStats,
    metricsToStatsResult,
    reducedStatsForFailure,
    statsReducer,
    statsResultToStatsSumResult,
} from "../../src/transformations/transformations";
import {ReducedStats, StatsResult, StatsSumResult} from "../../src/util/types";
import {GetMetricStatisticsOutput} from "aws-sdk/clients/cloudwatch";

describe('http result from stats', () => {
    test('should return a status code, headers and body', () => {
        const reducedStats: ReducedStats = {
            status: 'WARN',
            message: 'just a message',
        };

        const result = httpResultFromStats(reducedStats)

        expect(result.statusCode).toBe(200);
        expect(result.body)
            .toBe("{\"status\":\"WARN\",\"message\":\"just a message\"}");
        expect(result.headers['Access-Control-Allow-Origin']).toBe('*');
    });
});

describe('reduced stats for failure', () => {
    test('should return an err with the given stack name and message', () => {
        const result = reducedStatsForFailure('failure message');

        expect(result.status)
            .toBe('ERR');
        expect(result.message)
```

```
34              .toEqual("Could not retrieve info for this function. Error: failure mess\
35  age");
36      });
37  });
38
39  describe('metrics to stats result', () => {
40      test('should map function, metric and metric statistics to stats result', () => {
41          const data = {
42              Datapoints: [
43                  {
44                      Sum: 1
45                  },
46                  {
47                      Sum: 3
48                  },
49              ]
50          };
51
52          const result = metricsToStatsResult('aFunction', 'Invocations')
53              (data as GetMetricStatisticsOutput);
54
55          expect(result.functionName).toBe('aFunction');
56          expect(result.metric).toBe('Invocations');
57          expect(result.data).toEqual([{"Sum": 1}, {"Sum": 3}]);
58      });
59  });
60
61  describe('stats result to stats sum result', () => {
62      test('should change stats result to stats sum result', () => {
63          const data: StatsResult = {
64              functionName: 'aFunction',
65              metric: 'Invocations',
66              data: [
67                  {
68                      Sum: 1
69                  },
70                  {
71                      Sum: 3
72                  },
73              ]
74          };
75
76          const results = statsResultToStatsSumResult([data]);
```

```
77
78          const first = results[0];
79
80          expect(first.functionName).toBe('aFunction');
81          expect(first.metric).toBe('Invocations');
82          expect(first.sum).toBe(4);
83      });
84  });
85
86  describe('stats reducer', () => {
87      test('should return ok if there are invokes, few errors and no throttles', () =>\
88   {
89          const data: StatsSumResult[] = [
90              {
91                  functionName: 'first',
92                  metric: 'Invocations',
93                  sum: 70,
94              },
95              {
96                  functionName: 'second',
97                  metric: 'Errors',
98                  sum: 2,
99              },
100         ];
101
102         const result = statsReducer(data);
103
104         expect(result.status).toBe('OK');
105         expect(result.message).toEqual("invokes ok no errors");
106     });
107
108     test('should return warn if there are no invoke, errors or throttles', () => {
109         const data: StatsSumResult[] = [
110             {
111                 functionName: 'first',
112                 metric: 'Invocations',
113                 sum: 0,
114             },
115             {
116                 functionName: 'second',
117                 metric: 'Errors',
118                 sum: 0,
119             },
```

```
        ];

        const result = statsReducer(data);

        expect(result.status).toBe('WARN');
        expect(result.message).toEqual("not invoked no errors");
    });

    test('should return err if there are too many errors', () => {
        const data: StatsSumResult[] = [
            {
                functionName: 'second',
                metric: 'Errors',
                sum: 5,
            },
            {
                functionName: 'second',
                metric: 'Invocations',
                sum: 90,
            }
        ];

        const result = statsReducer(data);

        expect(result.status).toBe('ERR');
        expect(result.message).toEqual("throwing errors invokes ok");
    });

    test('should return warn if there are too many throttles', () => {
        const data: StatsSumResult[] = [
            {
                functionName: 'second',
                metric: 'Errors',
                sum: 3,
            },
            {
                functionName: 'second',
                metric: 'Invocations',
                sum: 75,
            },
            {
                functionName: 'second',
                metric: 'Throttles',
```

```
                sum: 2,
            }
        ];

        const result = statsReducer(data);

        expect(result.status).toBe('WARN');
        expect(result.message).toEqual("no errors invokes ok being throttled");
    });
});
```

If you now run `npm test`, jest will claim you have a test coverage of about 97%. Which is awesome, though you should know that jest only seems to take into account the files it 'touched' during the test run. So files that we did not write any tests for *and* are not used by our tested functions, are not included. But it is still great coverage for the files we did touch. To reach that kind of percentage in OOP, you have to write some really stupid tests (verifying the correctness of getters for example).

Add a folder called `util` under `__tests__` so you can write a test for our creation of `Between` types.

timeTest.ts

```
import {dateToBetween} from "../../src/util/time";

describe('date to between', () => {
    test('should give an end equal to given and a start three hours earlier', () => {
        const date = new Date(Date.UTC(2020, 10, 20, 10, 15, 30));

        const result = dateToBetween(date);

        expect(result.startTime.toISOString()).toBe('2020-11-20T07:15:30.000Z');
        expect(result.endTime.toISOString()).toBe('2020-11-20T10:15:30.000Z');
    });
});
```

That wasn't too bad. Ignoring calls.ts, we will show you how to write some tests for entrypoint (under the `metric-gateway` folder). We do need a little bit more setup for testing that file.

entrypointTest.ts

```
import {mocked} from 'ts-jest/utils';
import * as TE from "fp-ts/TaskEither";
import * as T from "fp-ts/Task";
import {GetMetricStatisticsOutput} from "aws-sdk/clients/cloudwatch";
import {getCWMetrics} from "../../src/metric-gateway/calls";
import {retrieveMetrics} from "../../src/metric-gateway/entrypoint";

jest.mock('../../src/metric-gateway/calls'); (1)

const getFromTaskEither = async (te: TE.TaskEither<string, StatsResult[]>) => {
    return TE.getOrElseW(() => T.task.of('failed'))(te)();
};

describe('retrieve metrics', () => {
    beforeEach( () => {
        mocked(getCWMetrics as any).mockClear();
    }); (2)

    test('should return metrics for a function', async () => {
        mocked(getCWMetrics as any)
            .mockImplementation(): Promise<GetMetricStatisticsOutput> => {
                return Promise.resolve({
                    Datapoints: [
                        {
                            Sum: 1,
                        }
                    ]
                })
        }); (2)

        const result: any = await getFromTaskEither(
            retrieveMetrics(
                {startTime: new Date(), endTime: new Date() },
                ['Invocations'])
            ('aFunction')
        ); (3)

        expect(result[0].functionName).toBe('aFunction');
        expect(result[0].metric).toBe('Invocations');
        expect(result[0].data).toEqual([{ Sum: 1} ])
    });
});
```

(1) This is important. We mock away the calls to AWS. The real calls have no place in a unit test.
(2) After clearing the mock in `beforeEach`, we make sure the call returns a Promise. Optionally, we could check the information we receive, but since our parameters have already been unit tested and our code is fairly simple, this will suffice.
(3) We use a helper we defined right after `jest.mock` to retrieve a result from the TaskEither, which we then check. If anything went wrong, we get back a Task with a string containing 'failed' and the assert will fail. Because the compiler does not know whether the result will be a string or a complex object, it will warn us unless we explicitly make `result` an `any` type. In production code we try to avoid the `any` type wherever we can, but in test code it's acceptable.

Even better would be to expand the above test suite with a test for a rejected Promise, but we will leave that 'as an exercise to the reader' (that's an annoying phrase, right?). If we now run `npm test` again, we see that jest thinks we have about 96% test coverage and both the time and entrypoint file are now included, though the index is still missing. Next time an architect or manager demands a test coverage of above 80%, just switch to a functional programming language!

Equally noteworthy is how low our cyclomatic complexity is. Put briefly, cyclomatic complexity measures the complexity of your program based on the number of paths within functions/methods. A function whose body just consists of a `return` has a complexity of 1. Adding `if/else` structures to such a function creates more possible paths and thus increases the complexity. A direct consequence of this increase in paths is an increase in the number of tests required to verify the function's behavior. With a single `if/else`, you already need at least two tests, because you have to verify both the if and else path. Some studies have also found a correlation between the cyclomatic complexity of programs and the number of bugs, which makes some intuitive sense. Every programmer knows how difficult it can be to read or write code with a lot of nested `if/elses`.

You can use tools like ESLint or the now deprecated TSLint to measure this. The default for the latter is a complexity of 20, which is a sensible limit for Object Oriented code. But for our application you might want to set the complexity to 2, *the lowest possible setting*. And you will find your linter warning you about only two functions, `statusMonoid` (complexity of 5) and - you may have guessed this one - `statsSumToStatusAndMessage` (complexity of 12). Our short, simple functions often only have one return and avoid `if/else` and `try/catch` structures, hopefully keeping most defects shallow.

Getting back on topic, I think the code in this chapter shows that small, pure functions are really easy to test. There are few possible paths which means often a single test is sufficient to cover almost all cases. No mocking is required either and test setup is minimal. I'm guessing that everyone who has worked in OOP has at some point tested a class that did way too many things (and had too many side effects) and therefore required tens or hundreds of lines of setup just to check a simple assertion. Most of our functions, on the other hand, only need data for their parameters. And testing our impure classes is harder, but doable using nothing more than the standard tools provided by a library like jest.

This is in part due to the way we organized our code, separating the impure from *the completely impure* stuff. Okay, that's a term I made up. With the latter I mean the actual impure functions whose only job is to do a call to some external system. With the former I mean functions that combine such

functions with pure functions. (Since we used Monads to wrap impure behavior, these are not, I guess, actually impure.) Anyway, the good news is that these *completely impure* functions - yes, there's that fake term again - are hardly worth *unit* testing! Typescript will point out basic mistakes and everything else will be covered by other types of tests. Because if you want any surity that your system will actually run, you definitely want to run your code against real systems, not just against your own mocks!

Property-based testing

Though in theory not limited to FP, property-based testing is popular within the FP community and largely unknown on the outside[16]. QuickCheck, the original library for this sort of testing, was written in Haskell and then ported to several other languages. Whereas unit testing will generally use the same few values to verify that your code is working, property-based testing generates a large number of random values with which to validate your code. This means your code is tested against a broad range of values, not the few you carefully selected, which is very good when your functions have to obey certain mathematical properties. When it comes to business code, however, the random nature of the values might limit its usefulness.

Let us look at an example. We have a function called product, which multiplies the values you give as parameters:

```
function product(first: number, second: number) {
    return first * second;
}
```

We want to be really sure that this function does what it is supposed to, and a large number of random values will give us additional assurances. Fp-ts does not have its own built-in quickcheck, but you can use fast-check[17] instead.

```
import * as fc from 'fast-check'
import { product } from './local/dir'

describe('product tests', () => {
    it('should multiply the given numbers', () => {
        const randomFirst: fc.Arbitrary<number> = fc.integer(); (1)
        const randomSecond: fc.Arbitrary<number> = fc.integer(); (1)

        fc.assert(
            fc.property(randomFirst, randomSecond, (first, second) => {
```

[16] Years ago our team used a Java library called RandomBeans (https://github.com/j-easy/easy-random) for automatically filling in fields of Java Objects, minimizing the amount of test setup we had to do. The difference lies in the goals. RandomBeans tries to minimize setup and repetitive work, whereas property-based testing is more interested in verifying laws.

[17] https://github.com/dubzzz/fast-check

```
                const result = product(first, second);

                expect(result).toBe(first * second);
            })
        ); (2)
    });
});
```

(1) Use a built-in function to generate two random integers.
(2) Now onto asserting. We pass in our random values and call `product` with them. In our `expect`, we compare the result with the product of the two values. These should be the same.

When run, this test succeeds. If we introduce a mistake in our product function, like adding a number after the multiplication, we get back something similar to *"Error: Property failed after 2 tests… Counterexample: [0,0]"*. Because we were adding a constant number, multiplying zero with zero no longer gives back the expected response. Getting back the value on which your function failed is extremely useful when you are generating random values. Obviously, if you run the code, the number of tests it took to find a failing example might be different due to the random nature of the produced values.

As said, the most obvious use case for property based testing is for code that has to obey (mathematical) laws, like product or sum. You can also verify things like sorting arrays. And of course many of the basic constructs of functional programming, like semigroups, monoids, monads… have laws as well. Which means property-based tests can verify that any monad you invent are in fact lawful. This probably goes a long way in explaining why this kind of testing is more prevalent in FP, with so many of its constructs coming from mathematics and category theory.

On the other hand, a lot of business rules are less easy to test with random values. As a simple example, how about a function that verifies that the given string is a valid email? Obviously, you cannot just pass in a random string. Because no matter how broad the category of valid email is, it still requires - at the very least - the presence of an @ sign. You could modify the random or create a custom one but even so, will you get any additional value from your test?

In other cases, property-based testing might be overkill. For example, say we have a function that takes a name and age and returns a `User`, without validation of any kind. We could pass in a random string and number, but would our test be better than one with hardcoded values? Likewise, if you have a function that takes only a limited amount of values you might be able to cover all relevant cases with a few unit tests.

So while property-based testing definitely has an important role to play in FP, it is especially important for verifying the correctness of basic components like those from fp-ts. For those of us who write business code on top of the components that others have kindly written for us, they are a useful addition. But often unit tests will be 'good enough'.

Other tools for better code

Besides tests, there are other tools you can use to make your code nicer and less buggy as well, like linters. Most Javascript projects use linters because they point out a lot of bad practices in Javascript code and help you fix them - often automatically! Eslint is one of the more common choices and can be used with Typescript as well. Some of its added value does disappear because of the partial overlap between compiler warnings and linter issues.

In addition, you could also add something like prettier which will automatically format your code, avoiding many pointless discussions within your team. Tools like type-coverage[18] will make sure your coverage remains above an agreed upon percentage, while `npm audit` will make sure your dependencies are kept up to date, etc. Whenever you think of a 'check' that might make your application better or safer, it is useful to Google whether someone has automated it yet. In many cases, the answer will be yes!

Summary

In this chapter we discussed the types of testing, TDD, the way unit tests work in FP and property-based testing. Turning from theory to practice, we showed how to write tests for our monitoring project. Yet again, we emphasized how small, pure functions make testing easier. We noted how easy it is to get to a high level of coverage, how mocks are not needed for pure functions and how keeping impure functions separate and limited means we have less need for (complicated) mocks.

[18] https://github.com/plantain-00/type-coverage

Once more unto the breach: our final application

Our final hands-on! We will again be writing a relatively complete application, this time with some more emphasis on TDD, communicating with the outside world and storing data.

Setting up

For this application, we will be handling hotel reservations. We will expand our range compared to the previous project, in that we will now have a REST API and will be saving data in a database. Everything will once again be deployed in the AWS Cloud.

I have no professional experience with the hotel business. So don't expect too much of my domain expertise.

We will leave a few things out of the picture though. We will not be handling user registration and authentication/authorization. Furthermore, we do not have the time to write a frontend/website for this project. That would only distract us from our goal, which is writing functional Typescript code, though it should be said that the React framework has a very functional feel to it. It encourages using pure functions and the creation of applications out of smaller components (like buttons and lists). So in spirit it will definitely play nice with a functional backend.

We will also be starting simple, expanding our project as we go. So we will start with an endpoint that receives reservation requests. Since we are changing data, the proper HTTP method for adding a reservation is POST. What data are we expecting? The obvious stuff: a hotel, a start and end date and the date of the reservation. The end date is the date that checkout is required, i.e. the room will be free again some time that same day. We will also receive a user id, which we will pretend has been sent along with the request by our authentication service. Our system will verify the received information, look up the hotel and make the reservation.

So let's do some setup. Obviously, we need a new empty project. We add a `package.json` in the root of the project:

package.json

```
{
  "name": "leanpub-reservations",
  "version": "1.0.0",
  "description": "Example reservations project with fp-ts and aws",
  "scripts": {
    "build-all-local": "lerna clean -y && lerna exec -- npm ci",
    "build-all": "lerna exec -- npm run build",
    "test-all": "lerna exec -- npm run test"
  },
  "dependencies": {},
  "devDependencies": {
    "jest": "^26.5.2",
    "ts-jest": "^26.4.1",
    "typescript": "^4.0.3",
    "ts-loader": "^5.3.3",
    "webpack": "^5.3.0",
    "webpack-cli": "^4.1.0",
    "lerna": "^3.22.1"
  },
  "author": "Sam Van Overmeire",
  "license": "ISC"
}
```

Some of this will look familiar. We added a whole bunch of dev dependencies that we will be needing now or later on. New among these is lerna[19]. Lerna is useful to us because we will have a separate package.json and setup for every Lambda. With the help of some lerna commands, we can run all the necessary commands for testing and building from the root of our project. As an example, consider the `build-all` command in the scripts section. It tells lerna to run `npm run build` in every folder with a package.json. You can specify which directory lerna should look for these folders, in a file called lerna.json (also located in the root of your project).

lerna.json

```
{
  "packages": [
    "lambdas/*"
  ],
  "version": "1.0.0"
}
```

So in our case, lerna has to check a directory called `lambdas`, which we will be creating shortly.

[19] https://github.com/lerna/lerna

With webpack we have an alternative way of building multiple projects: use entry-points[20] to create your output js files. You will still need custom code to zip these, but this is still a nice solution when you have a lot of Lambdas.

Because serverless is a relatively new field, standards around setting up projects are still a bit loose - especially compared to something like the conventional Maven setup with which every Java developer will be familiar. But generally, and this is language independent, you will want to separate Lambdas from each other. That makes it easier to keep your Lambdas focussed, your code clean and zips small - for faster startup times. For shared code, you can create utility folders.

We need a SAM template describing our infrastructure. So add the following:

infrastructure.yaml

```yaml
AWSTemplateFormatVersion: 2010-09-09
Transform: AWS::Serverless-2016-10-31

Parameters: (1)
  Environment:
    Type: String
    Default: dev

Globals: (2)
  Function:
    Runtime: nodejs12.x
    Handler: main.handler
    Environment:
      Variables:
        ENVIRONMENT: !Ref Environment

Resources:
  CreateLambda: (3)
    Type: AWS::Serverless::Function
    Properties:
      MemorySize: 512
      Timeout: 15
      CodeUri: lambdas/create/dist/create.zip
      Policies:
        - AWSLambdaExecute
      Events:
```

[20] https://webpack.js.org/concepts/entry-points/

```
27          Api:
28            Type: Api
29            Properties:
30              Path: /create
31              Method: GET
32              RestApiId: !Ref RestApi (4)
33
34    RestApi: (4)
35      Type: AWS::Serverless::Api
36      DeletionPolicy: "Retain"
37      Properties:
38        StageName: Prod
39
40  Outputs: (5)
41    ApiGatewayUrl:
42      Description: "API Prod stage endpoint"
43      Value: !Sub "https://${RestApi}.execute-api.${AWS::Region}.amazonaws.com/Prod/"
```

(1) We add a parameter to our infrastructure. `Parameters` are customizable values which have to be passed in when calling CloudFormation (though you can set default values - in this case the default is 'dev'). `Environment` is a common one, because it is often important to differentiate your environments (development, staging, preview, production...) from each other. Here this is done merely to show you the concept though, because we will only be deploying to one environment.
(2) `Globals` is a special SAM template section that help you avoid some duplication. In it, you can set values that will be applied to all functions within the template. So every Lambda will be running Node and will have a handler function within a main file. They will also have an environment variable called `ENVIRONMENT`. Because of `!Ref` (a so-called pseudo-parameter, already mentioned in the previous chapter) the value of that variable will be equal to whatever you pass in as the `Environment` parameter. So by default, it will be 'dev'.
(3) Most of this is similar to before. For now, the function only has basic permissions, like writing logs to Cloudwatch, thanks to the `AWSLambdaExecute` policy. The `Events` section is new. We saw before that this can be used to run our Lambda on a schedule. Here, however, we are telling AWS that we would like this Lambda to be triggered when there is a HTTP GET on the /create endpoint. *Note that this is for demonstration purposes only, later the method will change to an HTTP POST.*
(4) "Right", I hear you thinking, "but an endpoint of... what?" Ah, right. The Lambda is placed behind the endpoint of an API Gateway, which is an AWS managed service REST API. It creates an url for you and you tell it where to send traffic depending on method and path. So in our case, the only thing that will work is calling /create with GET, which will be routed to the Lambda we specified in (3). Again, note the `!Ref`. We want to tell `CreateLambda` the id of the API behind which it will be placed, using the `RestApiId`. We could create the API in a different template/stack and pass in or hardcode this id. But with `!Ref` we tell AWS to look at the resource with name `RestApi` and fill in its id.
(5) `Outputs` are values produced once the stack is created. Here we output the api gateway *base* url,

so we can use it to test our application. So if we want to call create, we will just take this output (which can be viewed in the CloudFormation console) and append "create".

If you are making a lot of mistakes writing CloudFormation templates and getting frustrated, you can either start using something more high-level, like the AWS CDK, or add tools like cfn-lint[21] which checks your templates for A LOT of mistakes (resource types that do not exist, invalid values, faulty use of pseudo-params, etc.). There are also CloudFormation plugins for IDEs like IntelliJ to give you some quick feedback.

One final thing for our project root, another bash script for weaving everything together.

setup.sh

```
#!/bin/bash

set -eo pipefail

STACKNAME="reservations-example"

if [ $# -lt 1 ]; then
    echo "Needs a bucket!"
    echo "Usage ./setup.sh <bucket>"
    exit 1
fi

BUCKET=$1

echo "Running build"
# npm run test-all => activate this when we actually have tests
npm run build-all

echo "Deploying stack"
aws cloudformation package \
--template-file infrastructure.yaml \
--s3-bucket "${BUCKET}" \
--output-template-file outputSamTemplate.yaml

aws cloudformation deploy \
--template-file outputSamTemplate.yaml \
--stack-name ${STACKNAME} \
--capabilities CAPABILITY_IAM

```

[21] https://github.com/aws-cloudformation/cfn-python-lint

```
echo "Removing temporary SAM file"
rm outputSamTemplate.yaml
```

Except for the stackname and `npm run build-all`, this is identical to our previous script. `npm run test-all` is commented out because we do not currently have any tests. In a professional environment you may want to look at using CodePipeline, Jenkins, CircleCI… to deploy your code. The AWS SAM CLI also has some commands to help you build and deploy serverless projects.

Now create a `lambdas` dir and, within that, a directory a `create` dir. Inside it, place the following:

package.json (in the create dir)

```
{
  "name": "leanpub-reservations-create-reservation",
  "version": "1.0.0",
  "description": "Lambda for creating a new reservation",
  "scripts": {
    "clean": "rm -rf node_modules && rm -rf dist",
    "webpack": "webpack",
    "zip": "cd dist && zip -r create.zip *",
    "build": "npm i && webpack && npm run zip",
    "test": "jest --coverage"
  },
  "dependencies": {
    "fp-ts": "^2.8.3"
  },
  "devDependencies": {
    "@types/jest": "^26.0.14"
  },
  "author": "Sam Van Overmeire",
  "license": "ISC"
}
```

The scripts are similar to the ones from a previous chapter, though this time lerna will be calling them, instead of us using them directly. Also similar to before, we need a jest, webpack and ts config.

jest.config.js

```
module.exports = {
    roots: [
        '<rootDir>',
    ],
    testMatch: [
        '**/__tests__/**/*.+(ts|tsx|js)',
    ],
    transform: {
        '^.+\\.(ts|tsx)$': 'ts-jest',
    },
};
```

webpack.config.js

```
const path = require("path");

module.exports = {
    resolve: {
        extensions: [".js", ".jsx", ".json", ".ts", ".tsx"],
    },
    output: {
        libraryTarget: "commonjs",
        path: path.join(__dirname, "dist"),
        filename: "[name].js",
    },
    target: "node",
    externals: ["aws-sdk"],
    module: {
        rules: [{ test: /\.tsx?$/, loader: "ts-loader" }],
    },
};
```

tsconfig.json

```
{
  "compilerOptions": {
    "lib": [
      "es2017",
      "esnext"
    ],
    "esModuleInterop": true,
    "moduleResolution": "node",
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "sourceMap": true,
    "target": "es2017",
    "outDir": "dist",
    "noImplicitThis": true,
    "noImplicitAny": true,
    "strictBindCallApply": true
  },
  "exclude": [
    "node_modules",
    "__tests__",
    "dist"
  ]
}
```

Especially important is the exclusion of the test folder. Else webpack will look at those files and complain about it not finding this Jest stuff you are talking about (since jest is a dev dependency).

Now create an `src` folder and place an index file in it:

index.ts

```
export const handler = async () => {
    return {
        statusCode: 200,
        body: JSON.stringify({ id: 'some-id' }),
    }
};
```

This will also be familiar, it is a very basic Lambda handler, which will be called by AWS whenever our `CreateLambda` is invoked. Just returning a "hello world" string is not enough: because a status code is lacking, anyone calling our endpoint would assume that something went wrong and throw an internal server error. So we return a http `statusCode` and put an example reservation id in the `body`, as a string.

Now you can run the setup script, with an S3 bucket as a parameter (see previously) and it will create a zip and deploy your application. If the deploy succeeds, you can find the base url of your new backend in the CloudFormation console. Or you could use the AWS CLI to retrieve it:

```
aws cloudformation describe-stacks \
--stack-name reservations-example \
--query "Stacks[0].Outputs[0].OutputValue"
```

This call will retrieve information from your stack and select the first (and only) output of the first (only) stack. Which should contain the url. If you use this url and append the /create path, you should get back an object containing a property `id` with value 'some-id'.

Writing our first Create Lambda code

This time we will use TDD for developing our application. For our very first code, we will be very explicit about the thought process. *Later on, this will become implicit* because this book would become way too long if we would build all our code incrementally, test by test.

The first thing we want to do when we receive an event in our Lambda, is to change it into a format we can actually work with and do some checks to see whether the information we received is valid (never trust frontend data). For example, we should check that the end date is later than the start date.

Create a folder under `create` called `__tests__`. Since we only have one function right now, we will put almost everything we write in the directory of the Create Lambda. If we notice we need certain functions elsewhere, in another Lambda, we will move them to a common module. So create a folder called `specifics` under our test folder. We will call our first test file `anticorruptionTest.ts`. In DDD, an anti corruption layer is often your first line of defence against the outside world. It can check incoming information and transform it into something that is actually useful to your domain. In our case, this layer will be modest. Add the following to the file.

anticorruptionTest.ts

```
import * as E from "fp-ts/Either";

const getResult = (either: any) => {
     return E.getOrElseW((err) => err)(either);
}

describe('anti corruption checks', () => {
    it('should fail when there is no start or end date', () => {
        const now = new Date();

```

```
        const eventData = {
            hotelId: '1',
            userId: '10',
            start: '',
            end: '',
            timestamp: now,
        }

        const event = {
            body: JSON.stringify(eventData)
        }

        const result = getResult(
             fromEventToCreateRequest(event)
        );

        expect(result).toBe('Start date should be before end date');
    });
});
```

In more recent years, developers have been advised to organise their code by component, not by layer. So instead of a module containing everything related to the controller (UserController, MoneyController) or service (UserService, MoneyService) or database, a module holds everything related to a certain concept in your domain (so UserController, UserService, UserDb all in one module). The reasoning behind this is that these components will change together, making modification of your code easier. Layers, by contrast, are easy to understand - but changes often require modifications to each layer, i.e. throughout your repository. In the right conditions, a Lambda project can encourage a component-based code organisation. You can create Lambdas focused on one concern within your domain (handling user interaction), with all the code they need (except obvious common functionality) inside their directory.

Since we placed our Lambda behind an API Gateway, we expect the incoming events to contain POST data. Which means everything we need is located in the body of the event. So we pass in our reservation data as a string inside the body. The function `fromEventToCreateRequest` will transform the event into a value we can use. It returns an Either because our checks might fail. So we use a helper to get the value from the Either. And in the case of our current test, we expect to find a string with an error message.

So let us create our `fromEventToCreateRequest` in `anticorruption.ts`:

anticorruption.ts

```
import * as E from 'fp-ts/Either';

export const fromEventToCreateRequest = (event: any) => E.right(event);
```

Now our test compiles, but fails. Before we turn it green, I'll add the types. So create a `types.d.ts` file. We will put this in a shared folder, so create a `util/domain` folder under the root of the project and add the file there.

types.d.ts

```
// REST
export type LambdaEvent = {
    body: string,
}

// DOMAIN
type CreateReservationRequest = {
    hotelId: string,
    userId: string,
    start: Date,
    end: Date,
    timestamp: Date,
};
```

Now we can change the types in our function.

anticorruption.ts

```
// ...

export const fromEventToCreateRequest = (event: LambdaEvent)
: CreateReservationRequest => E.right(event);
```

But that won't compile because we are returning the incoming event! So what can we write to make it compile? Perhaps something like this:

anticorruption.ts

```
export const fromEventToCreateRequest = (e: LambdaEvent): CreateReservationRequest =\
> E.right(undefined as CreateReservationRequest)
```

Now everything compiles just fine. But we still need to return the error message. So let us write a better implementation:

anticorruption.ts

```
import {CreateReservationRequest, LambdaEvent} from "../../../../util/domain/types";
import * as E from 'fp-ts/Either';
import {flow} from "fp-ts/function";

const toCreateReservationRequest = (event: LambdaEvent)
    : CreateReservationRequest => {
    console.log(event); // else typescript complains about not using the event
    return {} as undefined as CreateReservationRequest;
}

const checkCreateRequest = (createRequest: CreateReservationRequest) =>
createRequest.end > createRequest.start ?
        E.right(createRequest) :
        E.left('Start date should be before end date');

export const fromEventToCreateRequest = flow(
    toCreateReservationRequest,
    checkCreateRequest
);
```

If you dislike the long and ugly import for the domain util, take a look at Typescript paths[22] and tsconfig-paths[23]. Using paths you can shorten the import to `... from '@domain'`, which is quite nice.

If you run the test again, it will succeed. We already did some of the work we will need for the complete implementation. We have a pure function for changing the body into a create request, we have a `checkCreateRequest` that verifies the end date is after the start date and we have a function that composes these two using `flow`. Just like in earlier chapters.

Now write another test:

anticorruptionTest.ts

[22] https://www.typescriptlang.org/docs/handbook/module-resolution.html
[23] https://www.npmjs.com/package/tsconfig-paths

```
    // within the same 'describe' as before
    it('should return create request type when everything is ok', () => {
        const now = new Date();
        const oneDayInFuture = new Date(now.getTime() + 25 * 60 * 60 * 1000);

        const eventData = {
            hotelId: '1',
            userId: '10',
            start: now,
            end: oneDayInFuture,
            timestamp: now,
        }

        const event = {
            body: JSON.stringify(eventData)
        }

        const result = getResult(
            fromEventToCreateRequest(event)
        ) as CreateReservationRequest;

        expect(result.hotelId).toBe('1');
        expect(result.userId).toBe('10');
        expect(result.start instanceof Date).toBeTruthy();
        expect(result.end instanceof Date).toBeTruthy();
        // optionally more checks
    });
```

Now we get back an error again! So let us do some more work on our implementation:

anticorruption.ts

```
import {CreateReservationRequest, LambdaEvent} from "../../../../util/domain/types";
import * as E from 'fp-ts/Either';
import {flow} from "fp-ts/function";

const toCreateReservationRequest = (e: LambdaEvent): CreateReservationRequest =>
{
    const parsedBody: any = JSON.parse(e.body);

    return {
        hotelId: parsedBody.hotelId,
        userId: parsedBody.userId,
```

```
        start: parsedBody.start,
        end: parsedBody.end,
        timestamp: parsedBody.timestamp,
    };
}

const checkCreateRequest = (createRequest: CreateReservationRequest) =>
createRequest.end > createRequest.start ?
        E.right(createRequest) :
        E.left('Start date should be before end date');

export const fromEventToCreateRequest = flow(
    toCreateReservationRequest,
    checkCreateRequest
);
```

So we run our tests again and… our result start is not a Date? Ah, after the stringify it turned into a string. Ok, minor change:

anticorruption.ts

```
// imports

const toCreateReservationRequest = (e: LambdaEvent): CreateReservationRequest =>
{
    const parsedBody: any = JSON.parse(e.body);

    return {
        hotelId: parsedBody.hotelId,
        userId: parsedBody.userId,
        start: new Date(Date.parse(parsedBody.start)),
        end: new Date(Date.parse(parsedBody.end)),
        timestamp: parsedBody.timestamp,
    };
}

// other stuff is the same
```

Ok, so that works. Let us add one more test, with incorrect (but filled-in) dates:

```
    // other stuff

    it('should fail when the start date is after the end date', () => {
        const now = new Date();
        const oneHourInFuture = new Date(now.getTime() + 60 * 60 * 1000);

        const eventData = {
            hotelId: '1',
            userId: '10',
            start: oneHourInFuture,
            end: now,
            timestamp: now,
        }

        const event = {
            body: JSON.stringify(eventData)
        }

        const result = unsafeUnwrapLeft(
            fromEventToCreateRequest(event)
        );

        expect(result).toBe('Start date should be before end date');
    });
```

And everything still works, grand. But maybe you don't like writing a helper function to get the result from the Either function. Well, a companion library of fp-ts, fp-ts-std[24], has some helpers that start with `unsafe` to aid us:

```
import {fromEventToCreateRequest} from "../../src/specifics/anticorruption";
import {unsafeUnwrap, unsafeUnwrapLeft} from "fp-ts-std/Either";
import {CreateReservationRequest} from "../../../../util/domain/types";

describe('anti corruption checks', () => {
    it('should fail when there is no start or end date', () => {
        // same as before

        const result = unsafeUnwrapLeft(
            fromEventToCreateRequest(event)
        );

```

[24] https://samhh.github.io/fp-ts-std/docs/modules

```
13          expect(result).toBe('Start date should be before end date');
14      });
15
16      // the other test
17
18      it('should return create request type when everything is ok', () => {
19          // same as before
20
21          const result = unsafeUnwrap(
22              fromEventToCreateRequest(event)
23          ) as CreateReservationRequest;
24
25          expect(result.hotelId).toBe('1');
26          // more assertions
27      });
28  });
```

In Haskell and Purescript, functions starting with unsafe often throw an exception if anything goes wrong. The same is true here: `unsafeUnwrap` will throw an error if it encounters a `left`, `unsafeUnwrapLeft` will do so when it finds a `right`. Which is pretty stupid to do in production code (we are using Either specifically to *avoid* throwing exceptions) but is great in tests. There we actually want mistakes to blow up in our face.

Now let us consider some more minor improvements, this time to our production code. We are using `<` to compare our Dates. But fp-ts has built-in equals (Eq) and ordering (Ord) based on the equivalent in - you guessed it - Haskell. So we could use those instead.

anticorruption.ts

```
1  import {lt, ordDate} from "fp-ts/Ord";
2  // ...
3
4  const checkCreateRequest = (createRequest: CreateReservationRequest) =>
5      lt(ordDate)(createRequest.start, createRequest.end) ?
6          E.right(createRequest) :
7          E.left('Start date should be before end date');
```

Another thing you are probably also sick of hearing: the advantage of these fp-ts functions is that they can be combined, similar to how we created a new Monoid out of smaller parts. But Date is built-in, so no combining needed here.

After this change, our tests still run. Good. Our handling of the incoming body could be more 'honest' though. Parsing the JSON string might throw an error, so our signature would be more accurate if it returned an Either/Option/... So let us change that and demonstrate some more stuff from fp-ts while we are at it:

anticorruption.ts

```
// earlier imports
import {flip} from "fp-ts/function";
import {pipe} from "fp-ts/pipeable";
import {JsonRecord} from "fp-ts/lib/Either";
import {curry2} from "fp-ts-std/Function";

const getBody = (event: LambdaEvent) => event?.body; (1)
const parseJson = curry2(flip(E.parseJSON)); (2)

const toCreateReservationRequest = (body: JsonRecord): CreateReservationRequest => ({
    hotelId: body.hotelId as string,
    userId: body.userId as string,
    start: new Date(Date.parse(body.start as string)),
    end: new Date(Date.parse(body.end as string)),
    timestamp: new Date(Date.parse(body.timestamp as string)),
}); (3)

// check was not changed

export const fromEventToCreateRequest = (e: LambdaEvent): E.Either<string, CreateRes\
ervationRequest> =>
    pipe(
        getBody(e),
        parseJson(() => 'Could not parse body of request'),
        E.map(toCreateReservationRequest),
        E.chain(checkCreateRequest),
    ); (4)
```

(1) This function is just a helper that will return a body even if the event is undefined.
(2) This is a lot more interesting. The `parseJSON` function from fp-ts parses a string and returns an error message, which you specify as the second argument, in case of a failure. However, we would like to use `parseJSON` inside a `pipe` & we prefer point-free style. So we use `flip` to change the order of the arguments (placing the error message producer first) and then use `curry` to change the function into a curried version, so we can apply the arguments one by one. Inside our `pipe` we can then pass in the error message function. Passing in the actual string is implicit thanks to point-free style. Alternatively we could have 'prepared' our `parseJSON` further by immediately adding the message.

anticorruption.ts (alternative)

```
const parseJsonDefaultError = curry2(flip(E.parseJSON))
    (() => 'Could not parse body of request');

// ...
    pipe(
        getBody(event),
        parseJsonDefaultError,
        E.map(toCreateReservationRequest),
        E.chain(checkCreateRequest),
    );
```

(3) Anyway, we still keep everything we can in a pure function, `toCreateReservationRequest`. Note that because we get back a `JsonRecord`, we have to assert that the values inside are strings. Typescript knows they *could* be string, but they might also be another kind of valid JSON value (like a number or boolean). Our change also forces us to be more honest with timestamp, for which we want a date but are passing in a string, so we parse similar to start and end.
(4) Finally, our new exported function brings everything together using a `pipe` instead of `flow`. The `map` and `chain` should be familiar. Also note that Typescript is quite good at helping you out with the `flip`, `curry` and other helper functions.

To keep things simple, this is the only validation we will do. So we will ignore a check for the user id (which we assumed was going to be filled in by another one of our services), timestamp, hotel id (does it exist?), availability (are there rooms available at the given date?), etc. As we have seen in earlier chapters, we could easily write these checks independent from each other and combined them in our application. And, as before, we have a choice between Either or Validation.

Core logic of the Create Lambda

What we will do instead of writing these validations, is work on the core logic of our Create Lambda. Seems we have to do at least two things with the event we receive. One is, obviously, storing it somewhere. That is something we will be mocking for now, but we'll be coming back to that shortly. The other part is what we might again call *transformations*, modifying the data we have.

You may have noticed that there is no `price` included in the event. Because we do not completely trust the information coming from the frontend, we want to calculate the price in the backend, based on the dates we received. So we will start with that. Our `CreateReservationRequest` has now been confirmed to be valid by our validation, so we can use that as a starting point. We will also assume a flat rate of 20 euros for every night. So we add a file `priceCalculationTest.ts` to our `specifics` test folder:

priceCalculationTest.ts

```
import {calculatePrice} from "../../src/specifics/priceCalculation";

describe('price calculation', () => {
    it('should correctly calculate price for one night', () => {
        const startDate = new Date(2020, 11, 10, 12, 0, 0);
        const dayLater = new Date(2020, 11, 11, 12, 0, 0);

        const exampleCreateRequest: CreateReservationRequest = {
            hotelId: '1',
            userId: '10',
            start: startDate,
            end: dayLater,
            timestamp: undefined, // does not really matter here
        };

        const result = calculatePrice(exampleCreateRequest);

        expect(result.hotelId).toBe('1');
        // other checks for existing fields
        expect(result.price).toBe(20);
    });
});
```

Just to be clear, we wrote our code step by step again. First the test, next making sure it compiled and only then writing a real implementation. But for brevity's sake, this process is not shown. We create two dates, with a difference of 24 hours, put them in a request type and feed them to our new function. We expect to get back the values we sent *plus* a price equal to one night.

The calculator, added under `specifics` is quite simple:

priceCalculation.ts

```
// based on a StackOverflow answer
const differenceInDays = (start: Date, end: Date) => {
    return Math.floor(
        (new Date(end).getTime() - new Date(start).getTime())
        / 86400000
    );
}

export const calculatePrice = (r: CreateReservationRequest): CreateReservationPriced\
Request => {
    const days = differenceInDays(r.start, r.end);

```

```
    return {
        ...r,
        price: days * 20,
    };
};
```

`differenceInDays` is 'good enough for now'. With it we can calculate the number of days. Then we can add a `price` field with the number of days multiplied by the rate. And this rate could be extracted from the function to become a constant or an injected value.

You may have noticed we are returning a type that you have not seen before. We should add it to the types file.

types.d.ts

```
// DOMAIN
type CreateReservationPricedRequest = {
    hotelId: string,
    userId: string,
    start: Date,
    end: Date,
    timestamp: Date,
    price: number,
};
```

We discussed this approach in an earlier chapter: we prefer our program to be a linear flow with validations, transformations and effects. So now that we passed validation, we want to take our incoming request and change it into a new type with additional - or changed - information. Both for us (clarity, clear business logic) and our compiler (which knows what type is required at every step) this is better than adding optional fields to the original type, using boolean flags, etc.

You may have noticed that we do not require the entire Date, only the start and end *day*. And yet I did not just create a date with zero values for hours, minutes and seconds. This is a good (bad?) habit coming from experience with date errors. Like running tests locally and having them fail in a pipeline because the container was in a different timezone. These tests would (probably) have been OK, but since the hours do not matter, I still prefer to let them have a 'safe' value. Time-zone edge cases could probably be the subject of an entire book by itself! But it is not the subject of this one.

Though it is not strictly necessary, let's add another test for a stay of multiple days:

priceCalculationTest.ts

```
    // ...

    it('should correctly calculate price for three night', () => {
        const startDate = new Date(2020, 11, 10, 12, 0, 0);
        const dayLater = new Date(2020, 11, 13, 12, 0, 0);

        const exampleCreateRequest: CreateReservationRequest = {
            hotelId: '1',
            userId: '10',
            start: startDate,
            end: dayLater,
            timestamp: startDate, // does not really matter
        };

        const result = calculatePrice(exampleCreateRequest);

        expect(result.hotelId).toBe('1');
        expect(result.price).toBe(60);
    });
```

Ok, so that's enough for pricing right now. Now how about mock-saving to our database? Sure, let us start with a type:

types.d.ts

```
type CreateReservationPricedWithIdRequest = {
    hashKey: string,
    rangeKey: string,
    hotelId: string,
    userId: string,
    start: Date,
    end: Date,
    timestamp: Date,
    price: number,
}
```

The `hashKey` and `rangeKey` will be explained later. Suffice to say that they will be used by AWS DynamoDB and are an implementation detail. So it is arguable whether this should be added to this type, or hidden so it is not exposed to the rest of our application.

Now a test for creating values of the new type, in the `specifics` folder of our tests.

databaseTest.ts

```
import {addId} from "../../src/specifics/idGeneration";

describe('id generation', () => {
    it('should generate a primary key and sort key', () => {
        const startDate = new Date(2020, 11, 10, 12, 0, 0);
        const dayLater = new Date(2020, 11, 11, 12, 0, 0);

        const exampleRequest: CreateReservationPricedRequest = {
            hotelId: '1',
            userId: '10',
            start: startDate,
            end: dayLater,
            price: 20,
            timestamp: startDate,
        };

        const result = addId(exampleRequest);

        expect(result.hotelId).toBe('1');
        // ...
        expect(result.hashKey).toBe("RESERVATION#1");
        expect(result.rangeKey).toBe("2020-11-10#10");
    });
});
```

So again, a check that we get back at least one valid field that we passed in. And we expect the hash and range key to now be filled in. The values will be discussed when we write a real implementation. The preliminary implementation is:

database.ts

```
export const addId = (createRequest: CreateReservationPricedRequest): CreateReservat\
ionPricedWithIdRequest => ({
    hashKey: `RESERVATION#${createRequest.hotelId}`,
    rangeKey: `${createRequest.start.getUTCFullYear()}-${createRequest.start.getUTCM\
onth()}-${createRequest.start.getUTCDate()}#${createRequest.userId}`,
    ...createRequest
});
```

If the functions for hash and range key become more complex, we might consider splitting them off.

In typical fashion, getUTCMonth will do its best to confuse you and is therefore zero-indexed. So don't be surprised if you get range keys like '2020-0-10'.

Now we still want to mock-save our request and need a test for that as well. A save to a database is a typical call that will either return nothing, or nothing useful. So for now we prefer to just get back our (enriched) request.

databaseTest.ts

```
describe('save', () => {
    it('should save and then return an enriched event', async () => {
        const startDate = new Date(2020, 11, 10, 12, 0, 0);
        const dayLater = new Date(2020, 11, 11, 12, 0, 0);

        const exampleRequest: CreateReservationPricedRequest = {
            hotelId: '1',
            userId: '10',
            start: startDate,
            end: dayLater,
            price: 20,
            timestamp: undefined, // does not really matter here
        };

        // Task so () to turn it into a promise, then await
        const result = await saveToDatabase(exampleRequest)();

        expect(result.hotelId).toBe('1');
        expect(result.hashKey).toBe("RESERVATION#1");
        expect(result.rangeKey).toBe("START#2020-11-10#10");
    });
});
```

This is a **passthrough function**, something that executes a side effect and then returns what it received. It is also known as a **tee function** (like the unix command). There is one available for the IO monad in the fp-ts-std library where it is called `tap`. But our save will probably be asynchronous, so we use a different approach:

database.ts

```
const save = (createRequest: CreateReservationPricedWithIdRequest): T.Task<void> => {
    console.log(`Received ${JSON.stringify(createRequest)}`);
    return () => Promise.resolve();
};

export const saveToDatabase = (createRequest: CreateReservationPricedRequest)
    : T.Task<CreateReservationPricedWithIdRequest> => {
        return pipe(
            T.of(createRequest),
            T.map(addId),
            T.chainFirst(save),
        );
};
```

As usual, we kept our functions small, splitting off the function that has to perform a side effect (wrapped in a monad) and creating another function for combining it with others (like the pure `addId`). The only special thing is the use of `chainFirst`, which tells our code to ignore the output of `save` (which is `void`), but instead give us back what we passed to that function - i.e. the create request. Think of it like this: we have two Tasks to chain together (the one we already had is the first, the second is the one we get back from save) and the command literally says to only take the first chain - the first Task. So in effect the command ignores the Task we get back from `save` and our `saveToDatabase` acts like a passthrough.

Bringing it together

Two things are still missing. The obvious one is the index file that will glue everything together. Besides that, we also need a way to respond in a RESTful way when answering our callers. I.e. giving a status code and all that. So add this to our `specifics` folder:

responsesTest.ts

```
import {badRequestResponse, okResponse} from "../../src/specifics/responses";

describe('responses test', () => {
    it('should return a 200 response for ok response', () => {
        const result = okResponse('OK')

        expect(result.statusCode).toBe(200);
        expect(result.body).toBe('OK');
    });
});
```

Ok, that should be easy enough to implement.

responses.ts

```
import {Response} from "../../../../util/domain/types";

export const okResponse = (message: string): Response => ({
    statusCode: 200,
    body: message,
});
```

Note that we created a new type, which we add to our types:

types.d.ts

```
export type Response = {
    statusCode: number,
    body: string,
}
```

We also want to be able to report a failure, with a 400 message. So we add a test and implementation.

responsesTest.ts

```
    // ...

    it('should return a 400 response for bad request response', () => {
        const result = badRequestResponse('Error')

        expect(result.statusCode).toBe(400);
        expect(result.body).toBe('Error');
    });
```

responses.ts

```
export const badRequestResponse = (err: string): Response => ({
    statusCode: 400,
    body: err,
});
```

Now we are ready to combine all our moving parts. Let us write tests for the index.

indexTest.ts

```
import {handler} from "../src";

describe('create index test', () => {
    it('should return a 400 with message if start is later than end', async () => {
        const now = new Date();
        const oneHourInFuture = new Date(now.getTime() + 60 * 60 * 1000);

        const eventData = {
            hotelId: '1',
            userId: '11',
            start: oneHourInFuture,
            end: now,
            timestamp: now,
        }
        const event = {
            body: JSON.stringify(eventData),
        }

        const result = await handler(event);

        expect(result.statusCode).toBe(400);
        expect(result.body).toBe('Start date should be before end date');
    });

    it('should return 200 if everything is ok', async () => {
        const now = new Date();
        const oneHourInFuture = new Date(now.getTime() + 60 * 60 * 1000);

        const eventData = {
            hotelId: '1',
            userId: '11',
            start: now,
            end: oneHourInFuture,
            timestamp: now,
        }
        const event = {
            body: JSON.stringify(eventData),
        }

        const result = await handler(event);

        expect(result.statusCode).toBe(200);
        expect(result.body).toBe('Reservation created');
```

```
    });
});
```

As we mentioned earlier, it might become difficult to test our index file when we add more setup to it. But right now, we do not have any special configuration, so this will work. Now we implement our index file.

index.ts

```
import {LambdaEvent} from "../../../util/domain/types";
import {fromEventToCreateRequest} from "./specifics/anticorruption";
import {pipe} from "fp-ts/pipeable";
import * as E from 'fp-ts/Either';
import * as TE from 'fp-ts/TaskEither';
import {calculatePrice} from "./specifics/priceCalculation";
import {saveToDatabase} from "./specifics/database";
import {badRequestResponse, okResponse} from "./specifics/responses";

export const handler = async (event: LambdaEvent) => {
    return pipe(
        fromEventToCreateRequest(event),
        E.map(calculatePrice),
        TE.fromEither,
        TE.chain(r => TE.rightTask(saveToDatabase(r))),
        TE.map(() => okResponse('Reservation created')),
        TE.getOrElse((err) => () => Promise.resolve(badRequestResponse(err)))
    )();
};
```

None of this is new. While writing this code, I did consider various approaches for this piece of code. Like mapping from the Either to a Task and finally changing the left value in a Task as well. But using a TaskEither was as elegant and might be more suitable in the future - the possibility that we encounter an asynchronous Task that can fail is very real.

Still, some things could be nicer, like the `getOrElse` that seems to be doing a bit too much. So here is a slight refactor:

index.ts

```
// other imports
import * as T from 'fp-ts/Task';

export const handler = async (event: LambdaEvent) => {
    return pipe(
        fromEventToCreateRequest(event),
        E.map(calculatePrice),
        TE.fromEither,
        TE.chain(r => TE.rightTask(saveToDatabase(r))),
        TE.bimap(badRequestResponse, () => okResponse('Reservation created')),
        TE.getOrElse(T.of)
    )();
};
```

Instead of `map`, we have introduced `bimap`, which is very similar to the former but works with two values and requires two functions. In the case of (Task)Either, the first function receives the `left`, the second the `right`. And now we also use `T.of` to put our Response into a Task.

Everything compiles and the tests are green. You can now deploy and test this. You will have to change the "Method: GET" into a "Method: POST" in `infrastructure.yaml`. And since we are mocking the database, Cloudwatch logs might be useful in helping you verify that this really works.

Besides metrics, AWS Cloudwatch also has a logging service and Lambdas are configured to automatically send console logs there. The logs will only become visible *after* the first run of the Lambda, and they will be placed under `/aws/lambda/the-name-of-your-lambda`.

Creating the Retrieve Lambda

Next up: retrieving information about a reservation. How will that work? We will probably have to pass in two bits of information into the Lambda: the user id and hotel id. But that might return multiple results if a user has stayed in the hotel before. So we will also pass in the start date of the stay. And since we are retrieving information but not actually changing anything, the correct HTTP method is GET.

Create a new directory called `retrieve` under the lambdas directory. Copy the webpack config, jest config and tsconfig from the other lambda, those remain the same. The package.json is very similar:

package.json (retrieve lambda)

```
{
  "name": "leanpub-reservations-retrieve-reservation",
  "version": "1.0.0",
  "description": "Lambda for retrieving a reservation",
  "scripts": {
    "clean": "rm -rf node_modules && rm -rf dist",
    "webpack": "webpack",
    "zip": "cd dist && zip -r retrieve.zip *",
    "build": "npm i && webpack && npm run zip",
    "test": "jest --coverage"
  },
  "dependencies": {
    "fp-ts": "^2.8.3",
    "fp-ts-std": "^0.7.0"
  },
  "devDependencies": {
    "@types/jest": "^26.0.14"
  },
  "author": "Sam Van Overmeire",
  "license": "ISC"
}
```

There are of course ways to avoid (most of) this duplication. You can create a single webpack in the root of the project that does multiple builds, tsconfigs can inherit config from a 'base' config and you could do something similar for jest. But since we only have two Lambdas we will leave everything as-is, including this bit of duplication.

We again create an `src` folder with an index file. But this time, we will start with writing an index test, in the `__tests__` directory, which will only turn green when we have completed all our work for this Lambda:

indexTest.ts

```
import {handler} from "../src";

describe('retrieve index test', () => {
    it('should return the asked-for reservation', async () => {
        const startDate = new Date(2020, 11, 10, 12, 0, 0);

        const queryStringParameters = {
            hotelId: '1',
            userId: '11',
```

```
            start: startDate.toISOString(),
        }
        const event = {
            queryStringParameters,
        }

        const result: any = await handler(event);

        expect(result.statusCode).toBe(200);

        const body = JSON.parse(result.body);

        expect(body.price).toBe(20);
    });
});
```

Now add an index.ts with handler under the `src` folder to make it compile:

index.ts

```
import {LambdaEvent} from "../../../util/domain/types";

export const handler = async (event: LambdaEvent) => {
    return {
        statusCode: 200,
    }
};
```

The test will fail because there is no body present in the response. We will fix that eventually. Now that we have two Lambdas, we finally get some advantages from our use of lerna. We can now use `npm run test-all` and `build-all` to execute commands in both our lambda directories.

Let us also update our infrastructure:

infrastructure.yaml

```
  // ...

  RetrieveLambda:
    Type: AWS::Serverless::Function
    Properties:
      MemorySize: 512
      Timeout: 15
      CodeUri: lambdas/retrieve/dist/retrieve.zip
      Policies:
        - AWSLambdaExecute
      Events:
        Api:
          Type: Api
          Properties:
            Path: /retrieve
            Method: GET
            RestApiId: !Ref ServerlessRestApi
```

None of this is new. We add an additional resource, a Lambda function, with the same settings but a different `CodeUri`, `Path` and `Method`.

Now we can start working on the implementation. So add a folder called specifics to our `__tests__` and `src` folders and create an anticorruptionTest.ts.

anticorruptionTest.ts

```
import {unsafeUnwrap, unsafeUnwrapLeft} from "fp-ts-std/Either";
import {fromEventToRetrieveRequest} from "../../src/specifics/anticorruption";

describe('anti corruption', () => {
    it('returns a retrieve request for a valid payload', () => {
        const startDate = new Date(2020, 11, 10, 12, 0, 0);

        const queryStringParameters = {
            hotelId: '1',
            userId: '11',
            start: startDate.toISOString(),
        }
        const event = {
            queryStringParameters,
        }

        const result = unsafeUnwrap(fromEventToRetrieveRequest(event));

```

```
        expect(result.hotelId).toBe('1');
        // other checks
    });

    it('returns an error for a invalid payload', () => {
        const queryStringParameters = {
            hotelId: '1',
            userId: '11',
            start: '',
        }
        const event = {
            queryStringParameters,
        }

        const result = unsafeUnwrapLeft(fromEventToRetrieveRequest(event));

        expect(result).toBe('One or more empty fields. hotelId, userId and start are\
 required.');
    });
});
```

Since this is a GET, we will pass in useful information using the query parameters. These are passed in via the API Gateway event, so we can find where these are located by looking at examples, going through the AWS documentation or using the aws-lambda types repo[25].

types.d.ts

```
type QueryStringParams = {
    hotelId?: string,
    userId?: string,
    start?: string,
}

export type LambdaEvent = {
    queryStringParameters?: QueryStringParams,
    body?: string,
}

export type Reservation = {
    hotelId: string,
    userId: string,
    start: Date,
```

[25] https://www.npmjs.com/package/@types/aws-lambda

```
    end: Date,
    timestamp: Date,
    price: number,
};
```

The query strings are all optional, because they do not appear in our create Lambda. Similarly, we now made `body` optional with `?` because GET events do not have a body. I do hope that by now you instinctively consider an alternative approach: avoid optionals by creating two new types, a LambdaPostEvent and a LambdaGetEvent (or perhaps even more specific types). In which case: ten points to Gryffindor!

As seen in the test we just wrote, for a simple retrieval of a reservation we need three parameters. We will add a type `Reservation` for information that is sent back to the user. It is similar to the `CreateReservationPricedRequest` we already created, but since they serve very different goals it would be a mistake to try and avoid accidental duplication by using only one type. Right now our output is simply this Reservation type, though we could create a specific type for outputting to the outside world, a DTO. That does not seem needed at this point though.

Accidental duplication (as opposed to true duplication) is when you have code that is currently the same, but might grow in different directions in the future. And when you try to get rid of accidental duplication, you will run into trouble later on, when the two functions or pieces of code start drifting apart, because you will need special rules and additional code to deal with the needs of every function that uses the 'deduplicated' code. Say we eliminated the duplication between `CreateReservationPricedRequest` and `Reservation`. What if, in the future, we want to send a coupon with some `Reservations`? That field would also appear in our `CreateReservationPricedRequest` type where we do not need it. The situation would be even worse if we had some function for creating our new deduplicated `Reservation` - because we would have to make sure we do not accidently create a coupon when we are creating a reservation.

Ok, now our implementation:

anticorruption.ts

```
import * as E from 'fp-ts/lib/Either';
import {
    LambdaEvent,
    RetrieveReservationRequest
} from "../../../../util/domain/types";
import {pipe} from "fp-ts/pipeable";

const toRetrieveRequest = (event: LambdaEvent): RetrieveReservationRequest => ({
    hotelId: event.queryStringParameters?.hotelId,
    userId: event.queryStringParameters?.userId,
```

```
    start: new Date(Date.parse(event.queryStringParameters?.start as string)),
});

const checkRetrieveRequest = (event: LambdaEvent) => {
    return event.queryStringParameters?.hotelId
    && event.queryStringParameters?.userId
    && event.queryStringParameters?.start ?
        E.right(event) :
        E.left('One or more empty fields. hotelId, userId and start are required.')
};

export const fromEventToRetrieveRequest = (event: LambdaEvent): E.Either<string, Ret\
rieveReservationRequest> => {
    return pipe(
        checkRetrieveRequest(event),
        E.map(toRetrieveRequest),
    );
};
```

This is similar to what we did with our other Lambda. Note that we return a generic error message when a required parameter is missing. We have already seen how we could use Validation as an alternative. But is there a clean way to give back more specific messages while using Either? Well, besides composing Eithers for every specific error, which we might find to be too much work, for this use case we could also use guards.

A `guard` is another concept from the world of Haskell. It consists of several boolean checks, each coupled with a result, often with a final fallback in case none of the checks produce a match. A guard is included in the fp-ts-std library we already used earlier in this chapter and which is included in the dependencies of this Lambda.

anticorruption.ts

```
import {guard} from "fp-ts-std/Function";
// other imports and code

const checkRetrieveRequest = (event: LambdaEvent) => {
    const queryParamGuard = guard([
        [(e: LambdaEvent) => !e.queryStringParameters,
            () => `Missing query string params`],
        [(e: LambdaEvent) => !e.queryStringParameters.hotelId,
            () => `Missing hotel id`],
        [(e: LambdaEvent) => !e.queryStringParameters.userId,
            () => `Missing user id`],
        [(e: LambdaEvent) => !e.queryStringParameters.start,
```

```
            () => `Missing start`],
    ])(() => '');
    const message = queryParamGuard(event);

    return message ? E.left(message) : E.right(event);
};
```

What we did was import the guard and then add a number of nested arrays, with each nested array containing a function that evaluates to a boolean and a second function which returns a `string`. As a fallback, we return an empty string. Finally, if we have an error message, we return it wrapped inside a `left`, if not we return the event in a `right`. Note that our second test in `anticorruptionTest.ts` will now fail because the error message has changed, so that has to be changed.

The above code would be simpler if we could return a `left` for everything but the fallback, which could then return a `right` with the event. Unfortunately, this will not compile, because the guard was not written for use with **higher-kinded types** (and monads are such higher-kinded).

What about the database part? That's not a lot of work since we are still mocking the db with hardcoded data. Add a test and implementation:

databaseTest.ts

```
import * as TE from 'fp-ts/lib/TaskEither';
import * as T from 'fp-ts/lib/Task';
import {retrieve} from "../../src/specifics/database";

describe('retrieve from db', () => {
    it('should return the reservation if it exists', async () => {
        const request = {
            hotelId: '1',
            userId: '11',
            start: new Date(),
        }

        const result: any = await TE.getOrElse(
            (err) => T.of(err))(retrieve(request)
        )();

        expect(result.hotelId).toBe('1');
        expect(result.price).toBe(20);
    });

```

```
    it('should return an error message if it does not', async () => {
        const request = {
            hotelId: 'fake',
            userId: 'request',
            start: new Date(),
        }

        const result: any = await TE.getOrElse(
            (err) => T.of(err))(retrieve(request)
        )();

        expect(result).toBe('Unknown reservation')
    });
});
```

database.ts

```
import * as TE from 'fp-ts/lib/TaskEither';
import {
    Reservation,
    RetrieveReservationRequest
} from "../../../../util/domain/types";

const hardCode = {
    hotelId: '1',
    userId: '11',
    start: new Date(),
    end: new Date(),
    timestamp: new Date(),
    price: 20,
};

const db = (request: RetrieveReservationRequest) => {
    return request.hotelId === '1' && request.userId === '11' ? hardCode : undefined;
};

export const retrieve = (request: RetrieveReservationRequest): TE.TaskEither<string,\
 Reservation> => {
    return TE.tryCatch(() => {
        return new Promise((resolve, reject) => {
            const result = db(request);

            if(result) {
```

```
                return resolve(result);
            }
            return reject('Unknown reservation');
        });
    }, String)
};
```

We return a result for one very specific query. In other cases, we return an error message. What are we missing? Well: a way to give a nice, valid response. But we already wrote that for the other Lambda! And seeing as how the logic in there is very generic (take a string and put it inside an object with a status code), it seems safe to move this to a `common` folder, under `util`. To simplify things, just delete the `responsesTest.ts`. With a little bit of effort, we could also run tests for `util` with lerna, but we will skip that. The responses part will be tested indirectly by our index tests.

That means we can now bring everything together:

index.ts

```
import * as TE from 'fp-ts/lib/TaskEither';
import * as T from 'fp-ts/lib/Task';
import {LambdaEvent} from "../../../util/domain/types";
import {fromEventToRetrieveRequest} from "./specifics/anticorruption";
import {pipe} from "fp-ts/pipeable";
import {retrieve} from "./specifics/database";
import {badRequestResponse, okResponse} from "../../../util/common/responses";

export const handler = async (event: LambdaEvent) => {
    console.log(event);
    return pipe(
        fromEventToRetrieveRequest(event),
        TE.fromEither,
        TE.chain(retrieve),
        TE.map(res => okResponse(JSON.stringify(res))),
        TE.getOrElse((err) => T.of(badRequestResponse(err)))
    )();
};
```

A lot of imports, but not a lot of new code. The index test should now run, and we can expand it with one more test:

indexTest.ts

```
// ...

it('should return a 400 for an unknown reservation', async () => {
    const startDate = new Date(2020, 11, 10, 12, 0, 0);

    const queryStringParameters = {
        hotelId: 'false',
        userId: 'request',
        start: startDate.toISOString(),
    }
    const event = {
        queryStringParameters,
    }

    const result: any = await handler(event);

    expect(result.statusCode).toBe(400);
    expect(result.body).toBe('Unknown reservation');
});
```

And done. You can now deploy the application and test it with a browser, Postman or terminal commands. We can now, finally, turn to the database.

DynamoDB

Databases are a core component of any real-world applications and yet up until now we have been ignoring them. There are reasons for this: they complicate things while we want to focus on learning FP. And they should be kept out of the core part of your application, at the edges where they cannot contaminate other parts of your code. But a practical guide would be incomplete without integrating a database in at least one application. So we will be adding DynamoDB to ours.

DynamoDB is an AWS managed service, meaning there are no servers or anything for us to maintain. The only thing we specify is how much read and write capacity we want our database to have. In fact, even that is no longer required: DynamoDB can automatically scale capacity depending on traffic (so-called on-demand provisioning). You do pay a bit more for this convenience, so if your traffic is predictable, setting capacity yourself is still the better option.

But what kind of database is DynamoDB? It's a noSQL database of the key-value type, with a bit of the qualities of a document database mixed in. You specify an item by picking a single or composite key and adding any other attributes you desire. And those attributes can vary from item to item, or evolve with time. This is *schema-on-read*: you 'decide' what is in there when you retrieve the data (SQL databases on the contrary are generally *schema-on-write*, meaning you determine beforehand what a row can contain). Because you generally retrieve an item based on a key and writing is split

over a large number of partitions, such operations are very fast - milliseconds even at large scale. But your query options are limited to the keys as well as a limited number of optional indexes (Local Secondary Indexes or Global Secondary Indexes).

A popular notion, promoted by AWS itself, is single-table design: use one DynamoDB table to store all your data. This is done for reasons of efficiency both in cost and performance. Not only will your read/write be used more fully, with the right key mapping you may only need one query to retrieve various kinds of data, whereas you would obviously need several if your data was split over multiple tables. The downside is that you have to do even more planning when you decide on keys and indexes for DynamoDB. The core design question becomes "what kind of queries will I have to perform (efficiently)?".

In our project, we too will take a stab at using one table for all our information. You've already noticed how two keys, called `hashkey` and `rangekey`, appeared in our `addId` function. What are those? And why do they have those values? The hash key is the primary key. It is the only one that is strictly needed and must be unique if it appears alone. The range key is optional and offers a way to add multiple items with the same primary key. If you have a range key, it is the combination of hash and range that has to be unique.

Applied to our concrete case, we have created a hash key with the prefix 'RESERVATION#' followed by the hotel id. The range key is simply the date (year, month, day) on which the reservation starts plus the user id. This setup means that we can have multiple reservations per hotel and can ask our database to give us every reservation for a given hotel by passing in the hash key. The range key grants us some additional options. The obvious one is that we can find the reservation of a user, given his id and the start of his stay. But we can also ask to match a *starting substring* for the range key. Which means we can retrieve all reservations of a given date, a given year, etc.

Obviously, this particular setup has its limitations. We cannot use it to query all the reservations of a certain user, over every hotel (unless we want to 'scan' our entire database, often an antipattern). There are several solutions: get more creative (find a better way to build your keys), add additional (somewhat duplicated) data or use additional indexes.

What do I mean by adding additional data? As stated earlier, we can add multiple types of information inside one database. And at this point, the prefixes come in handy. We could add a USER# prefix, followed by the user id. That row could contain all the user info we have, both personal and regarding his reservations (saved inside the same row, or with references). In which case we could query with this new prefix to find information about all the user's reservations. Similarly, we could add a HOTEL# prefix followed by the hotel id to store data for the hotel that does not fit into the reservation category: name, number of rooms, etc. Another solution could be to just add a RESERVATION# followed by the user id (*instead of the hotel id*), with a range key that only contains the date. If we are using UUIDs, the chance of collision between keys would be minimal.

Note that all these examples mean we would be duplicating data! Something that is frowned upon in the world of relational databases with its stages of normalization. In the noSQL world, useful duplication is definitely allowed. We add some complexity in our code because we have to maintain multiple copies of the same data but we gain in database simplicity and performance. Even so, if

you really don't like the idea of duplicating, there are ways around it. We could add records for both the user and the hotel that only contain an id for the actual reservation info. Which would be mimicking foreign keys from relational databases, though without the security that those offer.

Better yet, we could turn to those Local Secondary Indexes (LSI) and Global Secondary Indexes (GSI). For our use case, we could add a GSI on the combination user id and range key. Once that's done, we can also retrieve reservations for a given user! There are disadvantages as well. Here too additional writes are required, but at least this time most of the logic is handled by AWS. You do have to make sure there is enough read/write capacity for these secondary indexes. Plus, you can only have a limited number of indexes, though AWS has increased the number significantly (you can have 20 GSI's now). So it is still advisable to think hard about your database keys setup and try to handle as many cases as possible with your hash and range key. When you find it impossible to handle each case, or new and unforeseen use cases appear on the radar, you can turn to indexes.

Adding the real database

Now we are ready to add the database to our system. First, we will update our infrastructure.

infrastructure.yaml

```
// ...

Globals:
  Function:
    Runtime: nodejs12.x
    Handler: main.handler
    Environment:
      Variables:
        ENVIRONMENT: !Ref Environment
        DATABASE_NAME: !Ref ReservationsDb (1)

Resources:
  CreateLambda:
    Type: AWS::Serverless::Function
    Properties:
      MemorySize: 512
      Timeout: 15
      CodeUri: lambdas/create/dist/create.zip
      Policies:
        - AWSLambdaExecute
        - Version: "2012-10-17" (2)
          Statement:
            - Effect: Allow
```

```
24                Action:
25                  - "dynamodb:*"
26                Resource:
27                  - "*"
28        Events:
29          Api:
30            Type: Api
31            Properties:
32              Path: /create
33              Method: POST
34              RestApiId: !Ref ServerlessRestApi
35    RetrieveLambda:
36      Type: AWS::Serverless::Function
37      Properties:
38        MemorySize: 512
39        Timeout: 15
40        CodeUri: lambdas/retrieve/dist/retrieve.zip
41        Policies:
42          - AWSLambdaExecute
43          - Version: "2012-10-17" (2)
44            Statement:
45              - Effect: Allow
46                Action:
47                  - "dynamodb:*"
48                Resource:
49                  - "*"
50        Events:
51          Api:
52            Type: Api
53            Properties:
54              Path: /retrieve
55              Method: GET
56              RestApiId: !Ref ServerlessRestApi
57
58  // rest of the file unchanged
```

(1) We now add another environment variable, which will inject the name of the database into the Lambdas.
(2) We give both our Lambdas permission to do anything they want with our DynamoDB databases. For better security, you can limit this to one resource (the specific database created in this file) and you can find out what DynamoDB calls you need. For example, our Create Lambda would need a `PutItem` permission.

Add the aws-sdk to the dev dependencies of both our Lambdas:

package.json (both create and retrieve lambda)

```
  "devDependencies": {
    "@types/jest": "^26.0.14",
    "aws-sdk": "2.828.0"
  }
```

While these are provided by AWS when running a Lambda (and do not have to be packaged with the rest of our dependencies), they are still needed and useful for local development.

Now to the code. The first thing we will do is create a new folder called `gateway`, similar to the naming of our previous project. Inside it, we create a file, similar to the previous chapter, a very simple call to an external service.

awsCalls.ts

```
import {DynamoDB} from "aws-sdk";
import {DocumentClient} from "aws-sdk/lib/dynamodb/document_client";
import PutItemInput = DocumentClient.PutItemInput;

const client = new DynamoDB.DocumentClient({region: 'eu-west-1'});

export const save = (params: PutItemInput) => {
    return client.put(params).promise();
};
```

Do be careful not to create the DocumentClient directly. If you do, Typescript will not be able to handle the DocumentClient type gracefully[26].

Now we can rewrite our databaseTest.

databaseTest.ts

[26] https://github.com/stevejhiggs/aws-lambda-typescript/issues/5

```
import {mocked} from 'ts-jest/utils';
import {addId, saveToDatabase} from "../../src/specifics/database";
import {CreateReservationPricedRequest} from "../../../../util/domain/types";
import {save} from "../../src/gateway/awsCalls";
import {DocumentClient} from "aws-sdk/lib/dynamodb/document_client";
import PutItemOutput = DocumentClient.PutItemOutput;

jest.mock("../../src/gateway/awsCalls");

describe('id generation', () => {
    // stays the same
});

describe('save', () => {
    beforeEach( () => {
        mocked(save as any).mockClear();
    });

    it('should save and then return an enriched event', async () => {
        const mockSave = mocked(save as any)
            .mockImplementation(():Promise<PutItemOutput> => {
                return Promise.resolve({});
            });

        const startDate = new Date(2020, 11, 10, 12, 0, 0);
        const dayLater = new Date(2020, 11, 11, 12, 0, 0);

        const exampleRequest: CreateReservationPricedRequest = {
            hotelId: '1',
            userId: '10',
            start: startDate,
            end: dayLater,
            price: 20,
            timestamp: startDate,
        };

        await saveToDatabase(exampleRequest)();

        expect(mockSave.mock.calls).toHaveLength(1);
    });
});
```

This is similar to the kind of testing we did earlier for our Cloudwatch calls. We mock the gateway

calls so we don't call any real external APIs. And to check that we are actually saving something, we verify that the mock was called once. We could (not shown here) further expand the test with assertions for the actual value of the save.

Now our database code:

database.ts

```
import * as TE from 'fp-ts/lib/TaskEither';
import {
    CreateReservationPricedRequest,
    CreateReservationPricedWithIdRequest
} from "../../../../util/domain/types";
import {flow} from "fp-ts/function";
import {DocumentClient} from "aws-sdk/lib/dynamodb/document_client";
import PutItemInput = DocumentClient.PutItemInput;
import {save} from "../gateway/awsCalls";

// addId is still the same

export const dynamoParams = (request: CreateReservationPricedWithIdRequest): PutItem\
Input => {
    return {
        TableName: process.env.DATABASE_NAME, // should move to config
        Item: {
            hashKey: request.hashKey,
            rangeKey: request.rangeKey,
            hotelId: request.hotelId,
            userId: request.userId,
            start: request.start.toISOString(),
            end: request.end.toISOString(),
            timestamp: request.timestamp.toISOString(),
            price: request.price,
        }
    };
} (1)

export const saveToDatabase = (createRequest: CreateReservationPricedRequest) => {
    const createParams = flow(addId, dynamoParams)(createRequest);
    return TE.tryCatch(() => save(createParams), String);
}; (2)
```

(1) We added a separate method for creating the parameters required by DynamoDB. It is a pure function and in a real project it should be tested either by itself, or with the additional assertions we

suggested earlier. Note that we are injecting the environment variable directly here. This is not ideal, because this is adding impurity to the otherwise pure function and it makes testing more difficult (we have to make sure to set the variable). We will work on this later.
(2) We already assumed we would be needing a TaskEither. And now we have moved to a `tryCatch` in order to handle any exceptions thrown by the external service. The rest of our setup has changed as well. Since we now start with two pure functions, we use `flow` to compose those, subsequently giving them to `save`.

For now we want to keep our index test, so we have to mock the calls to the gateway in a fashion similar to that of database test.

indexTest.ts

```
import {handler} from "../src";
import {save} from "../src/gateway/awsCalls";
import {mocked} from "ts-jest/utils";
import {DocumentClient} from "aws-sdk/lib/dynamodb/document_client";
import PutItemOutput = DocumentClient.PutItemOutput;

jest.mock("../src/gateway/awsCalls");

describe('create index test', () => {
    beforeEach( () => {
        mocked(save as any).mockClear();
    });

    it('should return a 400 with message if start is later than end', async () => {
        const mockSave = mocked(save as any)
            .mockImplementation(():Promise<PutItemOutput> => {
                return Promise.resolve({});
            });

        // other setup and test same as before

        expect(mockSave.mock.calls).toHaveLength(0);
    });

    it('should return 200 if everything is ok', async () => {
        const mockSave = mocked(save as any)
            .mockImplementation(():Promise<PutItemOutput> => {
                return Promise.resolve({});
            });

        // same as before
```

```

        expect(mockSave.mock.calls).toHaveLength(1);
    });
});
```

And our index file requires only one change.

index.ts

```
import {LambdaEvent} from "../../../util/domain/types";
import {fromEventToCreateRequest} from "./specifics/anticorruption";
import {pipe} from "fp-ts/pipeable";
import * as E from 'fp-ts/Either';
import * as T from 'fp-ts/Task';
import * as TE from 'fp-ts/TaskEither';
import {calculatePrice} from "./specifics/priceCalculation";
import {saveToDatabase} from "./specifics/database";
import {badRequestResponse, okResponse} from "../../../util/common/responses";

export const handler = async (event: LambdaEvent) => {
    return pipe(
        fromEventToCreateRequest(event),
        E.map(calculatePrice),
        TE.fromEither,
        TE.chain(saveToDatabase), (1)
        TE.bimap(badRequestResponse, () => okResponse('Reservation created')),
        TE.getOrElse(T.of)
    )();
};
```

(1) Before, we had a Task that needed to be wrapped in a TaskEither. Since that is no longer the case, just pass the function in - point free style!

That is it. Outside of the tests (and a lot of imports) our changes are limited. A few lines of code for setting up the external call, a function for building the parameters, `flow` instead of `pipe` (similar in complexity to what we had before) and a line in index that became simpler. Nice.

The only downside to adding it to our Retrieve Lambda as well, is that it is so similar. We already added permissions in our infrastructure and the aws-sdk is in the dev dependencies, so we can turn straight to the code. First, the Retrieve Lambda will also require a `gateway` folder with a call to AWS.

awsCalls.ts

```
import {DynamoDB} from "aws-sdk";
import {DocumentClient} from "aws-sdk/lib/dynamodb/document_client";
import GetItemInput = DocumentClient.GetItemInput;

const client = new DynamoDB.DocumentClient({region: 'eu-west-1'});

export const get = (params: GetItemInput) => {
    return client.get(params).promise();
};
```

This time, we are doing a `get`, which will require us to put a hash and range key in the params. But right now, the code for building that hash and range key is only available inside our create lambda! That won't do. Let us pull it out and put it in `util/common`, as dynamoSchema.ts:

dynamoSchema.ts

```
export const reservationHashKey = (hotelId: string) => `RESERVATION#${hotelId}`;
export const reservationRangeKey = (userId: string, start: Date) =>
    `${start.getUTCFullYear()}-${start.getUTCMonth()}-${start.getUTCDate()}#${userId\
}`

export const reservationKey = (hotelId: string, userId: string, start: Date) => ({
    hashKey: reservationHashKey(hotelId),
    rangeKey: reservationRangeKey(userId, start),
});
```

The first two functions are just extracted from `addId`. The last is a utility method, useful because in at least two cases we want both hash and range key. Usage in the create lambda becomes a tad different: use the spread operator for the reservation keys and add it to our object.

The spread operator . . . , which we have used once or twice before, has several nice qualities. It is flexible, concise (just the way Javascript likes it) and allows for an immutable way of creating new objects. The only downside is that copies are **shallow**.

database.ts (create)

```
import {reservationKey} from "../../../../util/common/dynamoSchema";
// other imports

export const addId = (createRequest: CreateReservationPricedRequest): CreateReservat\
ionPricedWithIdRequest => ({
    ...reservationKey(createRequest.hotelId, createRequest.userId, createRequest.sta\
rt),
    ...createRequest
});
```

Our retrieve database file has meanwhile lost some hardcoded values and a function, but gained some new ones:

database.ts (retrieve)

```
import * as TE from 'fp-ts/lib/TaskEither';
import * as E from 'fp-ts/lib/Either';
import {
    Reservation,
    RetrieveReservationRequest
} from "../../../../util/domain/types";
import {get} from "../../gateway/awsCalls";
import {DocumentClient} from "aws-sdk/lib/dynamodb/document_client";
import {pipe} from "fp-ts/pipeable";
import {reservationKey} from "../../../../util/common/dynamoSchema";
import GetItemOutput = DocumentClient.GetItemOutput;

const dynamoParams = (hotelId: string, userId: string, start: Date) => {
    return {
        TableName: process.env.DATABASE_NAME,
        Key: {
            ...reservationKey(hotelId, userId, start)
        }
    }
}; (1)

const fromItemToReservation = (output: GetItemOutput): Reservation => {
    const item = output.Item;

    return {
        hotelId: item['hotelId'],
        userId: item['userId'],
        start: item['start'],
```

```
        end: item['end'],
        timestamp: item['timestamp'],
        price: item['price'],
    };
}; (2)

const emptyCheck = (output: GetItemOutput) => output.Item ?
    O.some(output) : O.none; (3)

export const retrieve = (request: RetrieveReservationRequest): TE.TaskEither<string,\
 Reservation> => {
    return pipe(
        dynamoParams(request.hotelId, request.userId, request.start),
        params => TE.tryCatch(() => get(params), String),
        TE.map(emptyCheck),
        TE.map(
            O.map(fromItemToReservation)
        ),
        TE.chain(TE.fromOption(() => 'Unknown reservation'))
    )
}; (4)
```

(1) Another pure function for creating the parameters we need to pass to DynamoDB.
(2) Transformation from a DynamoDB `Item` to `Reservation`.
(3) This requires some DynamoDB knowledge. If no key is found, we will get back an empty object. So here we do check whether we have an `Item` with results. There are other options (no pun intended) for handling this, like wrapping `fromItemToReservation` with Either's `tryCatch`. But for missing data, Option seems the most appropriate pick.
(4) We bring everything together: we generate the params, pass them to TaskEither and inside the next two maps, we use Option and its map to handle what we got back from DynamoDB. Finally, we use `fromOption`, with a fallback, to get rid of the Option. Again, this one of several possible approaches.

No more changes to our production code are required, because what we are returning (a TaskEither with a potential `Reservation`) has not changed. So our changes are nice and contained. If you want to deploy the code, be sure to disable your index and database tests first. Even better, fix them! The code should be similar to that for the Create Lambda.

Summary

That covers our core functionality, which now includes a working database. We used a test-driven approach combined with already familiar techniques and fp-ts utilities to build this application. We

found that integrating a noSQL database was not that different from what we did in earlier chapters. We also saw that the impact of our changes was limited, sometimes in lines changed, in other cases because we only had to touch one file. In the next chapter, we will do some more work on this application and learn a few more things about fp-ts.

Expanding our reservation application

In this chapter, we consider some ways in which we can expand our basic-but-functional application. Some ideas will be implemented in their entirety, others will remain superficial sketches.

Using the Reader Monad for gathering config

Let us again consider the very unfortunate lack of configuration injection in some of our functions. When constructing parameters for retrieving or adding data to DynamoDB, we did this:

```
{
    TableName: process.env.DATABASE_NAME,
    // ...
}
```

I already mentioned that this is not ideal. And right now, only one environment variable in a limited number of places might seem OK. But what if we have to pass in a lot more configuration? The most basic solution would be to turn once again to currying, injecting information and using it later, when the function in question is called with its final argument(s). This works, but injecting manually can be a burden. Luckily, there is a monad for this, called **Reader**. You pass information to it and then retrieve it later on, when you need it. We'll demonstrate this technique within our Create Lambda.

First we need a function for generating config and some additional example configuration. Maybe the hotel with id '1', which we have been using in our tests, is a test object that we do not want to see in production. So we need to check which environment we are in. This is our type:

types.d.ts

```
export type Config = {
    tableName: string,
    environment: string,
    testHotelAllowed: boolean,
};
```

Besides the two environment variables, our config generates a boolean to verify whether test hotels are allowed. Another approach would be to add a list of test hotels to the configuration and check those in our validation.

Now add a config file in our `create/src/specifics` folder (of the Create Lambda):

config.ts

```
import {Config} from "../../../../util/domain/types";

export const generateConfig = (): Config => ({
    tableName: process.env.DATABASE_NAME,
    environment: process.env.ENVIRONMENT,
    testHotelAllowed: process.env.ENVIRONMENT !== 'prod',
});
```

Nothing too fancy. Our next steps are to add the config to those methods that need them. After which we will try to get everything to compile and fix our tests. We will start with our validation:

anticorruption.ts

```
import {
    Config,
    CreateReservationRequest,
    LambdaEvent
} from "../../../../util/domain/types";
// other stuff is still the same

const checkTestHotel = (c: Config) =>
    (createRequest: CreateReservationRequest) => {
        return c.testHotelAllowed || createRequest.hotelId !== '1' ?
            E.right(createRequest) :
            E.left(`Test hotel is not allowed in environment ${c.environment}`);
    }

export const fromEventToCreateRequest = (event: LambdaEvent) =>
    (config: Config): E.Either<string, CreateReservationRequest> =>
        pipe(
            getBody(event),
            parseJsonDefaultError,
            E.map(toCreateReservationRequest),
            E.chain(checkCreateRequest),
            E.chain(checkTestHotel(config)),
        );
```

We create a `checkTestHotel` function which will warn users if they are using the test hotel in the wrong environment. And we add it to our chain. Because we only expose `fromEventToCreateRequest` the configuration is injected here and passed on to the check method. One thing to note is that the config is the *second, not the first,* argument in `fromEventToCreateRequest`. In general, you should first inject things that will not change a lot - like configuration - saving whatever changes quickly (like data) for last. But for practical reasons, Reader expects your config/dependencies to be the last

argument. If you really dislike the signature, you could always write functions in the 'normal' order and then use a `flip` for use in combination with Reader. But in our case that is a lot of work for little reward.

Our database also requires the config:

database.ts

```
// other stuff has not changed

export const dynamoParams = (config: Config) =>
    (request: CreateReservationPricedWithIdRequest): PutItemInput => {
        return {
            TableName: config.tableName,
            Item: {
                hashKey: request.hashKey,
                // still the same
            }
        };
    }

export const saveToDatabase = (createRequest: CreateReservationPricedRequest) =>
    (config: Config) => {
        const createParams = flow(addId, dynamoParams(config))(createRequest);
        return TE.tryCatch(() => save(createParams), String);
    };
```

This is very similar to what we did for validation: add config to the exported function and pass it along. Now all we need to do is fix our index file:

index.ts

```
import * as RE from 'fp-ts/ReaderEither';
import * as RTE from 'fp-ts/ReaderTaskEither';
import * as RT from 'fp-ts/ReaderTask';
// the above three imports replace our Either, Task and TaskEither
import {generateConfig} from "./specifics/config";
// other imports unchanged

export const handler = async (event: LambdaEvent) => {
    const config = generateConfig(); (1)

    return pipe(
        RE.right(event), (2)
```

```
        RE.chain(fromEventToCreateRequest),
        RE.map(calculatePrice),
        RTE.fromReaderEither,
        RTE.chain(saveToDatabase),
        RTE.bimap(badRequestResponse, () => okResponse('Reservation created')),
        RTE.getOrElse(RT.of), (3)
    )
    (config) (4)
    (); (5)
};
```

(1) We generate our config.
(2) We place the event inside a ReaderEither - which is a Monad transformer, combining the qualities of Reader and Either - using `right`. That way we can `chain` the event to the ReaderEither generated by our next function.
(3) All the other functions remain the same, except we now use ReaderTask and ReaderTaskEither (a monad transformer consisting of three monads!). After the `getOrElse` you are left with a ReaderTask.
(4) Once this is done, we inject our config, which is passed to those functions that need it (and is ignored by the others). Which is also why we had to put that parameter in the ultimate position. So now we have a Task.
(5) And after the Task is called, it becomes a Promise. And we're done.

If you rewrite the above yourself, step by step & based on the earlier code, your compiler will complain until the very end. If you can't get it to work in one go, I find it useful to start with a temporary new `pipe` and add the functions gradually. It will compile and you can check what the pipe is outputting at any given moment.

This makes everything in our source code compile, except two of our test files. (The index test will still run: the output is the same and it was not paying attention to the table name anyway...) So we will fix our database and anticorruption tests, adding an additional test to the later file for our new functionality.

databaseTest.ts

```
// imports

jest.mock("../../src/gateway/awsCalls");

const config = {
    tableName: 'ourTable',
    environment: 'dev',
    testHotelAllowed: true,
}; (1)

describe('id generation', () => {
    // ...
});

describe('save', () => {
    beforeEach( () => {
        mocked(save as any).mockClear();
    });

    it('should save and then return an enriched event', async () => {
        // ...

        await saveToDatabase(exampleRequest)(config)(); (1)

        // ...
    });
});
```

(1) This is the only change: create a configuration and pass it to our function.

anticorruptionTest.ts

```
// imports

const config = {
    tableName: 'ourTable',
    environment: 'dev',
    testHotelAllowed: true,
}; (1)

describe('anti corruption checks', () => {
    it('should fail if test hotel is passed in when not allowed', () => {
        const noTestHotelConfig = {
```

```
12              tableName: 'ourTable',
13              environment: 'prod',
14              testHotelAllowed: false,
15          }; (2)
16
17          const now = new Date();
18          const oneDayInFuture = new Date(now.getTime() + 25 * 60 * 60 * 1000);
19
20          const eventData = {
21              hotelId: '1',
22              userId: '10',
23              start: now,
24              end: oneDayInFuture,
25              timestamp: now,
26          }
27
28          const event = {
29              body: JSON.stringify(eventData)
30          }
31
32          const result = unsafeUnwrapLeft(
33              fromEventToCreateRequest(event)(noTestHotelConfig)
34          );
35
36          expect(result).toBe('Test hotel is not allowed in environment prod');
37      });
38
39      it('should fail when there is no start or end date', () => {
40          // ...
41
42          const result = unsafeUnwrapLeft(
43              fromEventToCreateRequest(event)(config)
44          );
45
46          // ...
47      });
48
49      // same for the other two tests
50  });
```

(1) Again, create a configuration and pass it to the three existing tests.
(2) But also create an additional test with `testHotelAllowed` set to `false`. Since we are using our test hotel id, the check will fail and return an left.

That is all there is to it. We now have a nice, clean way of injecting a group of dependencies into our functions, using a monad. Gathering all these dependencies and values in one object is also a convenience. Instead of injecting numerous config parameters, only one additional parameter is required. And if you have different configs for different functions, there are solutions for this within Reader as well, using the `local` function. (But we won't be discussing a concrete example of that right now.)

Using State

I like using a Reader to pass on configuration to multiple functions. And the 'railway model' of always enriching our object with additional required information is a very nice way of thinking about functional programs. It means we can avoid keeping state within our application. I.e. we do not have to know that our request is now in the 'processed' state - that info is contained within the name/type/properties of the object! But in some cases, you will need to keep state. Which, in OOP, you would use your objects for. How do you go about doing that in a functional language? Enter the **State Monad**.

In essence, State provides a way for you to keep an object/value, which can be used/queried and changed by your function. A classic example for the use of State is producing a random value. The way you generate a random number in OOP languages is often to call a built-in method that outputs a number within a given range. Which means we have a method that, given identical parameters *does not produce the same output*. So we lose purity, which is not ideal. State is a way to solve this, because it requires you to pass a 'seed' value to a function. Depending on the seed, a different random value will be produced. Pass in the same seed and you will get the same value - saving purity.

In our case, we will improvise a reason - i.e not a very good one - to use State in our Retrieve Lambda. We will use this monad to check if the hotel id is our test hotel. If this is the case, we will reduce the price to `0`. That way, our testers do not have to worry about paying when they are just testing our code. If you don't like this reason, think of your own ad-hoc justification.

The core fp-ts library has several State flavors available, but the one we want (State + TaskEither) is not among them. For that, you will need to add the fp-ts-contrib package, another helper library, to your package.json:

```
1   "dependencies": {
2     "fp-ts": "^2.8.3",
3     "fp-ts-contrib": "^0.1.21",
4     "fp-ts-std": "^0.7.0"
5   }
```

Let us add a type for our State. We will call it `RetrieveState` to avoid name clashes with the monad. It contains an object with a single value, `isTest`, which indicates whether we are dealing with a test hotel or not.

types.d.ts

```
export type RetrieveState = {
    isTest: boolean,
};
```

In the anticorruption layer of our Retrieve Lambda we will add a function that will update this state if needed.

anticorruption.ts

```
export const checkHotelTest = (request: RetrieveReservationRequest)
    : STE.StateTaskEither<RetrieveState, unknown, RetrieveReservationRequest> =>
    (state: RetrieveState) => {
        const newState = request.hotelId === '1' ? {isTest: true} : state;
        return TE.of([request, newState]);
    };
```

There are two important things to note here. First, a StateTaskEither is simply a function that takes a State (in our case the `RetrieveState`) and returns a TaskEither. Second, a State is defined by not one, but two values: the initial one and the final one. Which means we have to return two things: the value we produced and a State. And that is exactly what we are doing in the TaskEither, which contains an array of our 'result' (the `RetrieveReservationRequest`) and the new State.

Returning the incoming state by default is only safe if we know the previous state had the property set to `false`. I mainly used it as a default here because else the compiler complains about me not using the incoming parameter (which I want to keep visible to show you that you need/can pass in the state). Alternatively, you can ignore the incoming State by removing the parameter. That will compile as well.

For our database we require two changes. First, the `fromItemToReservation` will return a 0 for `price` if the state is set to `test`. Second, the signature of our `retrieve` was changed. Similar to the above, it now is a function that takes a State and returns a TaskEither.

database.ts

```
// other imports
import * as STE from 'fp-ts-contrib/StateTaskEither';

// earlier code

const fromItemToReservation = (state: RetrieveState) =>
    (output: GetItemOutput): Reservation => {
        const item = output.Item;
        return {
            hotelId: item['hotelId'],
            userId: item['userId'],
            start: item['start'],
            end: item['end'],
            timestamp: item['timestamp'],
            price: state.isTest ? 0 : item['price'],
        };
    };

export const retrieve = (request: RetrieveReservationRequest): STE.StateTaskEither<R\
etrieveState, string, Reservation> =>
    (state: RetrieveState) => {
        const fromItemToReservationWithTestCheck = fromItemToReservation(state);

        return pipe(
            dynamoParams(request.hotelId, request.userId, request.start),
            params => TE.tryCatch(() => get(params), String),
            TE.map(emptyCheck),
            TE.map(
                O.map(fromItemToReservationWithTestCheck),
            ),
            TE.chain(TE.fromOption(() => 'Unknown reservation'))
            TE.map(r => [r, state]),
        )
    };
```

Now we can combine everything in our index. I've split up the pipe in two parts, to make it easier to read. The main difference is that the first pipe uses the `StateTaskEither`. It is a function that takes an initial state, where the `isTest` property has been set to `false`. Once the State has been applied, what we get back is a TaskEither. The next `pipe` has not changed - except now it maps on the first result of an array, since the second element is the state and is no longer needed. Nice to note: we are not doing any checks on the length of the array, because we are confident that we will in fact have two non-null elements to work with.

index.ts

```
// other imports
import * as STE from 'fp-ts-contrib/StateTaskEither';
import {
    checkHotelTest,
    fromEventToRetrieveRequest
} from "./specifics/anticorruption";

export const handler = async (event: LambdaEvent) => {
    const handleState = pipe(
        fromEventToRetrieveRequest(event),
        STE.fromEither,
        STE.chain(checkHotelTest),
        STE.chain(retrieve),
    )({isTest: false});

    return pipe(
        handleState,
        TE.map(res => okResponse(JSON.stringify(res[0]))),
        TE.getOrElse((err) => T.of(badRequestResponse(err)))
    )();
};
```

We disabled our tests earlier. Let's reactivate on of them so we can check that everything is working:

indexTest.ts

```
// other imports
import {mocked} from "ts-jest/utils";

jest.mock("../src/gateway/awsCalls");

describe('retrieve index test', () => {
    beforeEach( () => {
        mocked(get as any).mockClear();
    });

    it('should return the reservation with price 0 for test hotel', async () => {
        const startDate = new Date(2020, 11, 10, 12, 0, 0);

        const mockGet = mocked(get as any)
            .mockImplementation(): Promise<GetItemOutput> => {
                return Promise.resolve({
                    Item: {
```

```
                        hotelId: '1',
                        start: startDate.toISOString(),
                        end: startDate.toISOString(),
                        timestamp: startDate.toISOString(),
                        price: 100,
                    }
                });
            });

        const queryStringParameters = {
            hotelId: '1',
            userId: '11',
            start: startDate.toISOString(),
        }
        const event = {
            queryStringParameters,
        }

        const result: any = await handler(event);

        expect(result.statusCode).toBe(200);

        const body = JSON.parse(result.body);

        expect(body.price).toBe(0);
        expect(mockGet.mock.calls).toHaveLength(1);
    });

    // other tests
});
```

The jest setup should be familiar. Note that we are now asserting that the price is 0 for the test hotel, despite getting back a 100 from the database mock.

All in all, the hardest thing when it comes to working with State in fp-ts is learning how to create the proper function signature. So just remember, it is:

- a function that passes in a state
- returns the monad stack *minus* the state (so in our case TaskEither, for StateEither just an Either, etc.)
- contains an array containing your result and the state

Better error types

Another improvement: error handling. You may have noticed that we have, throughout this book, been putting our errors in strings. Which is okay for smaller projects, but strings are far from ideal. The reasons are probably familiar by now: strings are too generic and allow for a *very* wide range of information. So we know little about what will be contained inside them and our compiler is in the dark with us.

We will again use the Retrieve Lambda for this example. Let us first add some types:

types.d.ts

```
export type ClientError = {
    readonly type: 'ClientError',
    readonly message: string,
}
export type ServerError = {
    readonly type: 'ServerError',
    readonly message: string,
}

export type ErrorResponse = ClientError | ServerError
```

We've seen the above before, under various names. An error is either a `ClientError` or a `ServerError`, nothing else. It consists of a type (else we can't distinguish our errors) and a `message`. Everything is `readonly` - we generally do not want to change our message (and certainly not our type) once it's been created. We also need a way to build these errors, which we will add to our responses.

responses.ts (in common)

```
export const clientError = (message: string): ErrorResponse => {
    return {
        type: 'ClientError',
        message,
    };
}

export const serverError = (message: string): ErrorResponse => {
    return {
        type: 'ServerError',
        message,
    };
};

```

```

export const mapErrorToResponse = (err: ErrorResponse) => {
    return () => new Promise((resolve) => {
        switch (err.type) {
            case 'ClientError':
                return resolve(badRequestResponse(err.message));
            case 'ServerError':
                return resolve(internalServerErrorResponse(err.message));
            default:
                const _exhaustiveCheck: never = err;
                return _exhaustiveCheck;
        }
    });
};
```

The last function is the only interesting one. It provides a way to map our errors to what we will eventually give back to the user, in the shape of a HTTP response. If we have a client error, we will return a bad request (400). If it's a server error we are dealing with, it's an internal server error (500). Because we are using a sum type, we can once again use an exhaustiveness check to make our code more secure - forgetting to deal with a type of error will cause the compiler to complain.

We are using the generic terms 'clientError' and 'serverError' because we do not want REST related concepts littered throughout our code. The advantage here is that *in theory* if we want to move away from REST, the only thing we have to change is this responses file. Realistically speaking, the chance of us moving away from REST is small. And in REST, there is often one 'right' mapping for each type of error. A client error is definitely something in the 400 range. So we could get away with immediately mapping errors to http codes. Using non-specific terms is a nice touch and probably a best practice, but not vitally important.

Now the rest of our code changes should be small, though we will have to touch each file that throws errors plus the index file.

anticorruption.ts (again in Retrieve)

```
// other imports (and code)
import {
    ErrorResponse,
    LambdaEvent,
    RetrieveReservationRequest
} from "../../../../util/domain/types";
import {clientError} from "../../../../util/common/responses";

const checkRetrieveRequest = (event: LambdaEvent) => {
```

```
    const queryParamGuard = guard([
        // ...
    ])(() => '');
    const message = queryParamGuard(event);

    return message ? E.left(clientError(message)) : E.right(event); (1)
};

export const fromEventToRetrieveRequest = (event: LambdaEvent): E.Either<ErrorRespon\
se, RetrieveReservationRequest> => { (2)
    return pipe(
        checkRetrieveRequest(event),
        E.map(toRetrieveRequest),
    );
};
```

(1) We wrap our validation in client error, since a failure here clearly indicates a problem with the data we are receiving from the client.
(2) Our function now returns a `ErrorResponse` inside a `left`.

database.ts

```
// other imports (and code)
import {
    ErrorResponse,
    Reservation,
    RetrieveReservationRequest
} from "../../../../util/domain/types";
import {
    clientError,
    serverError
} from "../../../../util/common/responses";

export const retrieve = (request: RetrieveReservationRequest): STE.StateTaskEither<R\
etrieveState, ErrorResponse, Reservation> => (state: RetrieveState) => { (1)
    const fromItemToReservationWithTestCheck = fromItemToReservation(state);

    return pipe(
        dynamoParams(request.hotelId, request.userId, request.start),
        params => TE.tryCatch(() => get(params), (err: string) => serverError(err)),
        TE.map(emptyCheck),
        TE.map(
            O.map(fromItemToReservationWithTestCheck),
```

```
        ),
        TE.chain(TE.fromOption(() => clientError('Unknown reservation'))),
        TE.map(r => [r, state]),
    ) (2)
};
```

(1) Return an error response instead of a string.
(2) Inside the pipe we now throw a server error when our get fails. And when we do not find a reservation, we return a `clientError`.

index.ts

```
// other imports
import {mapErrorToResponse, okResponse} from "../../../util/common/responses";

export const handler = async (event: LambdaEvent) => {
    console.log(event);
    const handleState = pipe(
        fromEventToRetrieveRequest(event),
        STE.fromEither,
        STE.chain(checkHotelTest),
        STE.chain(retrieve),
    )({isTest: false});

    return pipe(
        handleState,
        TE.map(res => okResponse(JSON.stringify(res[0]))),
        TE.getOrElse(mapErrorToResponse) (1)
    )();
};
```

(1) This is the only required change. We now use the mapper we defined earlier to pick the right response for us. Handling errors that have been 'thrown' throughout our application is now a job for our `mapErrorToResponse`.

So now you can run your tests and verify… O right, we still have to fix a few of those. Let us do that and write an additional one, to placate the Test Gods.

anticorruptionTest.ts

```
// other imports
import {ClientError} from "../../../../util/domain/types";

describe('anti corruption', () => {
    // ...

    it('returns an error for a invalid payload', () => {
        const queryStringParameters = {
            hotelId: '1',
            userId: '11',
            start: '',
        }
        const event = {
            queryStringParameters,
        }

        const result = unsafeUnwrapLeft(
            fromEventToRetrieveRequest(event)
        ) as ClientError;

        expect(result.type).toBe('ClientError');
        expect(result.message).toBe('Missing start');
    });
});
```

We know this will be a client error, so we assert it is so and assert its type and message.

indexTest.ts

```
// imports

jest.mock("../src/gateway/awsCalls");

describe('retrieve index test', () => {
    // earlier code, unchanged

    it('should return a 400 for an unknown reservation', async () => {
        const mockGet = mocked(get as any)
            .mockImplementation((): Promise<GetItemOutput> => {
                return Promise.resolve({});
            });

        const startDate = new Date(2020, 11, 10, 12, 0, 0);
```

```

        const queryStringParameters = {
            hotelId: 'false',
            userId: 'request',
            start: startDate.toISOString(),
        }
        const event = {
            queryStringParameters,
        }

        const result: any = await handler(event);

        expect(result.statusCode).toBe(400);
        expect(result.body).toBe('Unknown reservation');
        expect(mockGet.mock.calls).toHaveLength(1);
    });

    it('should return a 500 for a server error reservation', async () => {
        const mockGet = mocked(get as any)
            .mockImplementation(): Promise<GetItemOutput> => {
                return Promise.reject('reason');
            });

        const startDate = new Date(2020, 11, 10, 12, 0, 0);

        const queryStringParameters = {
            hotelId: 'false',
            userId: 'request',
            start: startDate.toISOString(),
        }
        const event = {
            queryStringParameters,
        }

        const result: any = await handler(event);

        expect(result.statusCode).toBe(500);
        expect(result.body).toBe('reason');
        expect(mockGet.mock.calls).toHaveLength(1);
    });
});
```

Little of this is new. We mock with jest and make our mock return empty or throw an error, to

simulate a bad request and internal server error. And now we can be quite sure that our code does what we expect.

All in all, it's a nice extension of our current approach to errors. In our types.d file, we have a nice overview of every type of error we expect in our domain, which acts as living documentation. And type signatures still act as documentation of where errors might be thrown. By using custom errors we have also made our compiler more powerful. It can now give us more guidance compared to when we were using strings. Finally, instead of throwing exceptions haphazardly we return them as values when needed and handle them at the end of our program, in the `getOrElse`, in a single place/function.

In our introduction, we mentioned how libraries like morphic-ts[27] can reduce the amount of boilerplate that using ADTs in Typescript entails. So before continuing, let's demonstrate that. With morphic, you can either use the library to create ADTs with a lot of convenience methods, or you can pass in existing types/interfaces to build a somewhat simpler ADT. We will do the latter for our error responses.

Installing morphic (in the root of your project) might give an error about a missing fast-check peer dependency. If that is the case, just run `npm i fast-check@1.21.0` before/in combination with `npm i morphic-ts` and everything should be fine.

responses.ts

```
import {Response, ErrorResponse, ClientError, ServerError} from "../domain/types";
import {makeADT, ofType} from "morphic-ts/lib/adt"; // new imports

// other code unchanged

export const MorphicErrorResponse = makeADT('type')({
    ClientError: ofType<ClientError>(),
    ServerError: ofType<ServerError>(),
}); (1)

export const mapErrorToResponse = (err: ErrorResponse) => {
    return () => MorphicErrorResponse.match({
        ClientError: ({message}) => {
            return Promise.resolve(badRequestResponse(message))
        },
        ServerError: ({message}) => {
            return Promise.resolve(internalServerErrorResponse(message))
        },
    })(err);
}; (2)
```

[27] https://github.com/sledorze/morphic-ts

(1) We call `makeADT` to build an Algebraic Data Type from our two error types.
(2) Now we use the `match` method of the resulting `MorphicErrorResponse` to rewrite our error mapper. Compared to a switch this requires less ceremony and comes closer to the pattern matching of Haskell. We specify what needs to happen for each type of `MorphicErrorResponse` (client error and server error) and get direct access to its properties, i.e. `type` and `message`. Here we only need the latter, which we pass to our `Response` functions. As a minor variation on the previous version of our error mapper, we now use `Promise.resolve` instead of `new Promise`.

So `match` gives us more elegant, readable code. As a bonus, it will freak out when you do not handle all types - whereas with a switch we had to provide a default case that IntelliJ disliked and wanted to get rid of. Morphic provides additional conveniences, like methods to instantiate types. Which means we can remove the `clientError` and `serverError` functions and instead use `of.ClientError` and `of.ServerError`:

anticorruption.ts

```
import {MorphicErrorResponse} from "../../../../util/common/responses";
// other imports and code

const checkRetrieveRequest = (event: LambdaEvent) => {
    // guard, unchanged

    return message ?
        E.left(MorphicErrorResponse.of.ClientError({ message })) (1)
        : E.right(event);
};
```

database.ts

```
import {MorphicErrorResponse} from "../../../../util/common/responses";
// other imports and code

export const retrieve = (request: RetrieveReservationRequest): STE.StateTaskEither<R\
etrieveState, ErrorResponse, Reservation> => (state: RetrieveState) => {
    const fromItemToReservationWithTestCheck = fromItemToReservation(state);

    return pipe(
        hotelParams(request.hotelId, request.userId, request.start),
        params => TE.tryCatch(() => get(params), (message: string) => {
            return MorphicErrorResponse.of.ServerError({message}); (1)
        }),
        TE.map(emptyCheck),
        TE.map(
            O.map(fromItemToReservationWithTestCheck),
```

```
        ),
        TE.chain(TE.fromOption(() => {
            return MorphicErrorResponse.of.ClientError({message: 'Unknown...'}); (1)
        })),
        TE.map(r => [r, state]),
    );
};
```

(1) Since `message` is the only property besides the type - which the library fills in for us - it is the only required parameter.

These generated constructors are quite simple and do not provide any validation, so if you need that you still have to write your own smart constructors. Morphic offers a lot of other utilities besides those demonstrated here, including reducers, transformers (for immutably modifying objects) and lenses. Really the only downside is the introduction of an additional library and everything that that can entail.

Restaurants and sequenceS

Now consider another scenario. When someone makes a reservation for a hotel, he can also make a reservation for a restaurant at the same time. When we call our Retrieve Lambda, we expect to get back both the hotel and optionally the restaurant. And yes, this is another scenario that just serves as an excuse to show something from fp-ts. Sorry.

In that case, we would have to change our Retrieve to get both these items, preferably in parallel. We will suppose our type for a restaurant reservation looks like this:

types.d.ts

```
export type RestaurantReservation = {
    hotelId: string,
    userId: string,
    start: Date,
    timestamp: Date,
    price: number,
    freeDrink: string,
};
```

It is quite similar to a hotel reservation, except there is a free drink string, indicating what drink you get as a reward for also booking the restaurant. How thoughtful. We also need a primary key:

dynamoSchema.ts

```
export const restaurantReservationHashKey = (hotelId: string) =>
`RESTAURANT#${hotelId}`;

export const restaurantReservationKey = (hotelId: string, userId: string, start: Dat\
e) => ({
    hashKey: restaurantReservationHashKey(hotelId),
    rangeKey: reservationRangeKey(userId, start),
});
```

So the hash key or primary key uses the RESTAURANT# prefix followed by the hotel id. The range key is the same as before.

database.ts

```
// same imports as before

// name change
const hotelParams = (hotelId: string, userId: string, start: Date) => {
    // same
};

// name change
const fromItemToHotelReservation = (state: RetrieveState) => (output: GetItemOutput)\
: HotelReservation => {
    // same
};

const restaurantParams = (hotelId: string, userId: string, start: Date) => {
    return {
        TableName: process.env.DATABASE_NAME,
        Key: {
            ...restaurantReservationKey(hotelId, userId, start)
        }
    };
};

const fromItemToRestaurantReservation = (output: GetItemOutput): RestaurantReservati\
on => {
    const item = output.Item;
    return {
        hotelId: item['hotelId'],
        userId: item['userId'],
        start: item['start'],
```

```
        end: item['end'],
        timestamp: item['timestamp'],
        price: item['price'],
        freeDrink: item['freeDrink'],
    };
};

const emptyCheck = (output: GetItemOutput) => output.Item ? O.some(output) : O.none;

// name change
export const retrieveHotel = (request: RetrieveReservationRequest)
    : STE.StateTaskEither<RetrieveState, ErrorResponse, HotelReservation> => (state:\
 RetrieveState) => {
    // same
};

export const retrieveRestaurant = (request: RetrieveReservationRequest)
    : STE.StateTaskEither<RetrieveState, ErrorResponse, O.Option<RestaurantReservati\
on>> => (state: RetrieveState) => {
    return pipe(
        restaurantParams(request.hotelId, request.userId, request.start),
        params => TE.tryCatch(() => get(params), (err: string) => serverError(err)),
        TE.map(emptyCheck),
        TE.map(
            O.map(fromItemToRestaurantReservation),
        ),
        TE.map(r => [r, state]),
    );
}
```

Our database setup is quite similar to the one for hotel. In fact, there is some duplication that we could get rid of - but that might be too soon. It might yet prove to be accidental duplication. One difference is that our `retrieveRestaurant` returns an Option, because it is perfectly acceptable if we do not find a restaurant for every reservation. Now onto our index:

index.ts

```
// other imports
import * as O from 'fp-ts/lib/Option';
import {sequenceS} from "fp-ts/Apply";

const ado = sequenceS(STE.stateTaskEither); (1)

export const handler = async (event: LambdaEvent) => {
    console.log(event);

    const handleState = pipe(
        fromEventToRetrieveRequest(event),
        STE.fromEither,
        STE.chain(checkHotelTest),
        STE.chain(r =>
            ado({
                hotel: retrieveHotel(r),
                restaurant: retrieveRestaurant(r),
            })), (1)
    )({isTest: false});

    return pipe(
        handleState,
        TE.map((res) => {
            const reservations = res[0];
            return JSON.stringify({
                hotel: reservations.hotel,
                restaurant: O.getOrElse(() => null)(reservations.restaurant),
            }); (2)
        }),
        TE.map(okResponse),
        TE.getOrElse(mapErrorToResponse)
    )();
};
```

(1) Earlier, we used `sequenceT` to do two parallel tasks and get back a tuple. Here, we prefer an object, so we use `sequenceS` (the S stands for 'structure'). After the example of others, including the creator of fp-ts, we call it ado. Why that name? In Haskell, `do` offers some syntactic sugar for working with monads. `ado` does the same for **Applicative** values. After defining it, we use it inside a chain to create an object with `hotel` and `restaurant` properties. Which we pass along to the next pipe.
(2) Now, instead of just creating a string from the first element of the array, which is what we did before, we add a bit more logic to replace the `none` with an empty element. Optionals are primarily for use *inside* our application. Outside of it, we are perfectly fine with using null or undefined. Other applications will hopefully also have some anti corruption layer for translating unsafe outside values.

Yet *another* alternative would have been to use **These** - which is discussed briefly in the appendix.

We should also change our index test:

indexTest.ts

```
// ...

    it('should return the reservation with price 0 for test hotel', async () => {
        const startDate = new Date(2020, 11, 10, 12, 0, 0);

        const mockGet = mocked(get as any)
            .mockImplementation((params: GetItemInput): Promise<GetItemOutput> => {
                if (params.Key['hashKey'] === 'RESERVATION#1') {
                    return Promise.resolve({
                        Item: {
                            hotelId: '1',
                            start: startDate.toISOString(),
                            end: startDate.toISOString(),
                            timestamp: startDate.toISOString(),
                            price: 100,
                        }
                    });
                }
                return Promise.resolve({});
        });

        const queryStringParameters = {
            hotelId: '1',
            userId: '11',
            start: startDate.toISOString(),
        }
        const event = {
            queryStringParameters,
        }

        const result: any = await handler(event);

        expect(result.statusCode).toBe(200);

        const body = JSON.parse(result.body);

```

```
        expect(body.hotel.price).toBe(0);
        expect(body.restaurant).toBeNull();
        expect(mockGet.mock.calls).toHaveLength(2);
    });
```

Because we are now doing two get calls, we will check the incoming hash key and only return a result when asked for a hotel reservation. And in our asserts, we check that the hotel is still present, that the restaurant is null and get has been called twice. This should now run, and the other tests should work without changes.

Choosing configs

We dealt with a tiny configuration in our Create Lambda. But often applications not only have much bigger configs, but also various defaults to fall back to. You might have default program settings that are overwritten by system settings, overwritten by specific user settings. Are there elegant ways to handle such a situation, preferably without too much conditional branching? For easy scenarios, you might consider using **Alt** and **Alternative**. Alt is similar to semigroup, which we discussed way earlier in this book, except that it can work with monads. Alternative is Alt expanded with an identity element. So similar to how monoid added an empty value to semigroup.

Take a look at this example.

```
import * as O from 'fp-ts/Option';

type Conf = {
    environment: string,
}

const basic: Conf = {
    environment: 'dev',
}

const advanced: Conf = {
    environment: 'prod',
}

const mostAdvanced: Conf = {
    environment: 'superprod'
}

const basicFallback = () => O.some(basic);
const advancedFallback = () => O.some(advanced);
```

```
21 const emptyFallback = () => O.none;
22
23 const advancedConfig = O.alt(basicFallback)
24     (O.alt(advancedFallback)(O.some(mostAdvanced)));
25
26 const prodFallbackConfig = O.alt(basicFallback)
27     (O.alt(advancedFallback)
28     (O.zero()));
29
30 const basicFallbackConfig = O.alt(basicFallback)
31     (O.alt(emptyFallback)
32     (O.zero()));
```

We have three kinds of configuration. The most basic one is always available, but the others might not exist. So when we retrieve our config, we use `alt` and specify `basic` as the first fallback and the advanced fallback as a second alternative. If the most advanced configuration is available, which is the case in `advancedConfig`, it is selected. In `prodFallbackConfig` it is `zero` (which is the identity value for Option, passing in `O.none` gives the same result), so we get back the advanced configuration. If this one is also unavailable, as it is in `basicFallbackConfig`, we get back our most basic config, with our dev environment.

But I personally like this example[28] of how to use Monoids for config/settings even better! Let us apply it to our Reader Lambda configuration:

```
1  import { Monoid, getStructMonoid } from 'fp-ts/lib/Monoid'
2  import * as O from 'fp-ts/lib/Option'
3
4  type Config = {
5      tableName: O.Option<string>,
6      environment: O.Option<string>,
7      testHotelAllowed: O.Option<boolean>,
8  };
9
10 const basicConfig: Config = {
11     tableName: O.some('test-db'),
12     environment: O.some('dev'),
13     testHotelAllowed: O.some(false),
14 };
15
16 const advancedConfig: Config = {
17     tableName: O.fromNullable(process.env.DATABASE_NAME),
18     environment: O.fromNullable(process.env.ENVIRONMENT),
```

[28] https://dev.to/gcanti/getting-started-with-fp-ts-monoid-ja0

```
    testHotelAllowed: O.fromNullable(process.env.ENVIRONMENT !== 'prod'),
};

const monoidSettings: Monoid<Config> = getStructMonoid({
    tableName: O.getLastMonoid<string>(),
    environment: O.getLastMonoid<string>(),
    testHotelAllowed: O.getLastMonoid<boolean>()
});

const config = monoidSettings.concat(basicConfig, advancedConfig);
```

Our config now has optional fields and we created a basic configuration containing some sensible values. We also created our 'real' config and wrapped the environment variables in `fromNullable`, so if they are missing, we will get back empty Options. Next, we create a monoid for `Config` and tell it to take the *last* monoid it can find for every individual property. Finally, we combine our basic and advanced configurations.

Note that we put the basic config in first. This is important. Since we said to get the last Monoid, fp-ts will start looking at the advanced config first. So if there is a filled-in Option for `tableName` in the advanced config, it will be found first and selected. If it is a `none`, the monoid proceeds to the left and finds the filled in basic table name. This happens for every property. So any filled-in environment variable will become part of our config, those that are empty are replaced by the values from the basic config. So what is nice about this is that selection happens at the property level and we can easily create more complex configuration setups, as long as we have a clear precedence. For example basic settings < system settings < user settings gives us:

```
const config = monoidSettings.concat(basicSettings, systemSettings, userSettings);
```

And if you prefer to work from left to right, there is a `getFirstMonoid`.

Exploring some more ideas

Before going to the summary, we will briefly discuss some other possible expansions.

How about making the price depend on the hotel? What would change is that you would get another type of data in the database, probably with the prefix HOTEL#, containing all relevant information about a hotel, like its name, location and pricing. We would also have to change code in the Create Lambda: before calculating the price, we would do a retrieval of price data from the database, meaning our Either would become a TaskEither a bit earlier than before. And to pass the information along, we could enrich our `CreateReservationRequest` with price information (`roomPrice` for example), while leaving `CreateReservationPricedRequest` as is because when the request has been priced, we no longer need that information. As usual, having the right kind of information in our data structure at the right time will help our compiler help us. So this would be

a change in quite a few files (index, database, priceCalculation), though almost all of it contained within the specifics of the Create Lambda.

We could also make it possible to get discounts for early booking, perhaps depending on the hotel - in which case we might save this information in the database, maybe under HOTEL#. The rest of the scenario is also similar to above: retrieve this information, add it to the type and expand the price calculation. If we have already done the above, this new feature will not affect the composition in our index, limiting the scope of our changes.

What if we had new restrictions on hotels? Maybe some are only for adults, or limited to certain nationalities? That's a task of our anticorruption file, though impact on our composition would depend on the way this information is retrieved. If it is from a database (from the hotel info?) our validation would become asynchronous.

How would we handle 'follow-up events', by which I mean things like emails confirming your reservation? One way would be to add this to the workflow in our Create Lambda. But in a cloud environment, other approaches might be more appropriate for reasons both of speed and modularity. Speed can be important to a user who wants to know that his reservation was successfully made, preferably before the browser times out. And Lambdas like to do 'one thing well', so just adding the sending of emails for no good reason does not make sense.

Thus, this change would require work on the architectural level. One option would be to use DynamoDB Streams, which allows you to configure a Lambda to listen to events from the database. Like, say, the creation of a new reservation. At which point your new function can send a confirmation email. Such an approach fits in well with the idea of **event sourcing**, which - briefly put - puts events front and center, with events consisting of anything that happened within your domain. In our case we could have a ReservationRequestEvent (a user wants a reservation), a ReservationConfirmedEvent (the hotel has confirmed the reservation), ReservationDeniedEvent, ReservationCancelledEvent, and so on. These are immutably saved within a database, which is your **single source of truth**. If you do want to 'change' an event, you would do this by taking compensatory action - undoing an event with a new one: a ReservationConfirmedEvent followed by a ReservationCancelledEvent.

Your entire application will evolve around creating these events *and* reacting to them. After a ReservationRequestEvent, code would run checking whether the reservation is actually possible (in event sourcing parlance this would be a 'command') and if so, generate a ReservationConfirmedEvent. Next, code which is listening to this type of event sends an email to the customer. Which is very similar to what we proposed before: listen to the creation of reservations and use that info to send an email. The result is a decoupled, modular system where the user gets speedy initial confirmation, followed by an email at a later stage. The downside is a more complex system, with a lot of tiny moving parts.

Another option would be to use AWS Step Functions. Step Functions help coordinate different Lambdas. They can keep state, pass information from one Lambda to another, wait for additional input, etc. In our case the Step Function would be responsible for taking action to ensure the reservation is successfully handled, including informing the user. Mind you, in the simple example

we presented earlier, where we are only sending an email, this is overkill. On the other hand, if there are a lot of follow-up steps, a Step Function might be simpler than message-driven communication between Lambdas, even if the latter will probably be faster and definitely cheaper. For this topic in general, search for 'orchestration versus choreography'.

Summary

In this chapter, we looked at some ways to expand our reservation application. We learned about some more monads (Reader and State) along the way, how we can use Alt or monoids for picking the right settings and saw `sequenceS` in action. You should now have a good toolkit for tackling most programming problems in functional Typescript.

Rewriting an application

This chapter is based on a series of blog posts[29] which I wrote to show a) the downsides of writing an application in an hybrid imperative functional/OOP style and b) how to rewrite it. The text has been modified to make it similar to the rest of this book.

The original application

In this chapter we deviate from our normal approach and take as our starting point an already written application. It is called **Pet Service**, and is written in an imperative style using both objects and functions. But despite some advantages (you use functions/objects when and where they are suitable), there are numerous issues with this coding approach, which we will explore here.

Pet Service is a CRUD application that stores information about pets and wild animals. Pets have more attributes than wild animals (like cuteness), but for the latter we save a counter for each type of animal, because it might be nice to know just how many lions are out there. For both preservation and safety purposes.

The infrastructure setup is quite similar to what we have been doing in earlier chapters. Our code sits behind an API Gateway HTTP API (a variation on the kind of API Gateway we used previously but faster, cheaper and a bit simpler) which sends every call to a greedy endpoint, a (fat) Lambda that takes care of further routing. Once again, DynamoDB serves as the database.

The root of the application directory contains the IaC, a deploy script and `package.json`. The `package.json` contains lambda-api as a dependency. Lambda-api handles routing of the incoming events. It is quite similar to libraries like express and serverless-express.

package.json

```
{
  "name": "petservice",
  "version": "1.0.0",
  "scripts": {
    "build": "rm -rf dist node_modules; mkdir dist; npm i --production && zip dist/p\
ets.zip -r src -r node_modules"
  },
  "license": "ISC",
  "dependencies": {
    "lambda-api": "0.10.6"
```

[29] https://medium.com/codex/making-an-existing-application-more-functional-4e1ed8b53131

```
  },
  "devDependencies": {
    "aws-sdk": "^2.764.0"
  }
}
```

The setup script is about the same as the ones we've been using in previous chapters.

setup.sh

```
#!/bin/bash

set -eo pipefail

STACKNAME="pets-example"

if [ $# -lt 1 ]; then
    echo "Needs a bucket!"
    echo "Usage ./setup.sh <bucket>"
    exit 1
fi

BUCKET=$1

echo "Running build"
npm run build

echo "Deploying stack"

aws cloudformation package \
--template-file infrastructure.yaml \
--s3-bucket "${BUCKET}" \
--s3-prefix backendbuilds \
--output-template-file outputSamTemplate.yaml

aws cloudformation deploy \
--template-file outputSamTemplate.yaml \
--stack-name ${STACKNAME} --capabilities CAPABILITY_IAM

echo "Removing temporary SAM file"
rm outputSamTemplate.yaml && rm -rf dist
```

And our infrastructure should look familiar as well, except perhaps for the HTTP API part.

infrastructure.yaml

```yaml
AWSTemplateFormatVersion: 2010-09-09
Transform: AWS::Serverless-2016-10-31

Parameters:
  Environment:
    Type: String
    Default: dev

Resources:
  DefaultLambda:
    Type: AWS::Serverless::Function
    DependsOn: LambdaLogGroup
    Properties:
      FunctionName: !Ref AWS::StackName
      CodeUri: dist/pets.zip
      Runtime:  nodejs14.x
      Timeout: 10
      MemorySize: 1024
      Handler: src/index.handler
      Role: !GetAtt LambdaFunctionRole.Arn
      Environment:
        Variables:
          DATABASE_NAME: !Ref Database
          ENVIRONMENT: !Ref Environment

  LambdaFunctionRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: 2012-10-17
        Statement:
          - Effect: Allow
            Principal:
              Service:
                - lambda.amazonaws.com
            Action:
              - sts:AssumeRole
      ManagedPolicyArns:
        - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
      Policies:
        - PolicyName: !Sub "${AWS::StackName}-lambda-role"
          PolicyDocument:
            Statement:
```

```
44                - Effect: Allow
45                  Action:
46                    - dynamodb:*
47                  Resource:
48                    - "*"
49
50    HttpApi:
51      Type: AWS::ApiGatewayV2::Api
52      Properties:
53        Name: !Sub ${AWS::StackName}
54        Description: Example Http Api For Pet Service
55        ProtocolType: HTTP
56        Target: !Sub
57          - arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-31/functions/${lambd\
58  aArn}/invocations
59          - lambdaArn: !GetAtt DefaultLambda.Arn
60
61    LambdaPermission:
62      Type: AWS::Lambda::Permission
63      Properties:
64        Action: lambda:InvokeFunction
65        FunctionName: !Ref DefaultLambda
66        Principal: apigateway.amazonaws.com
67        SourceArn: !Sub "arn:aws:execute-api:${AWS::Region}:${AWS::AccountId}:${HttpAp\
68  i}/*/$default"
69
70    LambdaLogGroup:
71      Type: AWS::Logs::LogGroup
72      Properties:
73        LogGroupName: !Sub "/aws/lambda/${AWS::StackName}"
74        RetentionInDays: 30
75
76    Database:
77      Type: AWS::DynamoDB::Table
78      Properties:
79        TableName: !Sub "${AWS::StackName}-database"
80        BillingMode: PAY_PER_REQUEST
81        AttributeDefinitions:
82          - AttributeName: ppId
83            AttributeType: S
84          - AttributeName: psId
85            AttributeType: S
86        KeySchema:
```

```
        - AttributeName: ppId
          KeyType: HASH
        - AttributeName: psId
          KeyType: RANGE
```

All application code is located in the `src` folder. As usual, an index file serves as an entrypoint to our lambda.

index.js

```
const api = require('lambda-api')({
    logger: {
        level: 'debug',
        access: true,
        stack: true,
        serializers: {
            req: (req) => ({
                query: req.query,
                params: req.params,
            }),
        },
    },
});
const routes = require('./routing');

routes.apply(api);

exports.handler = async (event, context) => {
    console.log(event);
    // lambda-api issues with newer event versions
    if(event.version === "2.0") {
        event.httpMethod = event.requestContext.http.method;
        event.path = event.requestContext.http.path;
    }
    return api.run(event, context);
};
```

Routing is configured inside the `routing.js` file. It makes sure incoming requests are validated, sets up POST and GET calls and auto-generates OPTIONS endpoints.

routing.js

```
const { bodyValidation, queryParamValidation } = require('./lib/helpers');
const { callFor } = require('./lib/items');
const { Pet } = require('./lib/pets');
const { WildAnimal } = require('./lib/wildAnimals');

function routesToMapOfPathsWithAvailableMethods(currentRoutes) {
    const optionRoutes = {};
    currentRoutes.forEach((route) => {
        (optionRoutes[route[1]] = optionRoutes[route[1]] || []).push(route[0]);
    });
    return optionRoutes;
}

function generateOptionCalls(api) {
    const currentRoutes = api.routes();
    const optionRoutes = routesToMapOfPathsWithAvailableMethods(currentRoutes);

    Object.keys(optionRoutes).forEach((key) => {
        api.options(key, (request, response) => {
            response.header(
                'Access-Control-Allow-Methods',
                `${optionRoutes[key].join(',')},OPTIONS`,
            );
            response.sendStatus(200);
        });
    });
}

function apply(api) {
    const pets = callFor(Pet);
    const wild = callFor(WildAnimal);

    api.use(queryParamValidation);
    api.use(bodyValidation);

    api.get('/pets/:id/client/:clientId', pets.get);
    api.post('/pets/:id/client/:clientId', pets.post);
    api.get('/wild/:id/type/:type', wild.get);
    api.post('/wild/:id/type/:type', wild.post);

    generateOptionCalls(api);
}

```

```
module.exports = { apply };
```

`config.js` contains some constants, including the DynamoDB table name which is retrieved from the Lambda environment variables.

config.js

```
const CONSTANTS = {
    PETS_ID: 'P',
    WILD_ANIMALS_ID: 'W',
    COUNTER_ID: 'C',
};

module.exports = {
    ...CONSTANTS,
    DDB_TABLE: process.env.DATABASE_NAME,
};
```

All other files are under the lib folder. `helpers.js` contains functions for validation; `pets.js` and `wildAnimals.js` have classes describing both types of animal. Each class groups its attributes and can generate keys for storage in or retrieval from the database.

helpers.js

```
function validateString(value) {
    if (!value.trim()) throw new Error('Received an empty value');
}

function validateObject(obj) {
    if (typeof obj === 'undefined') return;
    if (typeof obj === 'object') {
        Object.keys(obj).forEach(key => {
            validateString(key);
            const type = typeof obj[key];

            if (type === 'function') {
                throw new Error('No functions allowed');
            }
        });
    }
}

function bodyValidation(request, response, next) {
    if (!request.body && request.method !== 'POST') {
```

```
21          next();
22      } else {
23          try {
24              validateObject(request.body);
25              next();
26          } catch (e) {
27              console.error(`body validation error: ${e}`);
28              return response.sendStatus(400);
29          }
30      }
31  }
32
33  function queryParamValidation(request, response, next) {
34      if (!request.query) {
35          next();
36      } else {
37          try {
38              Object.keys(request.query).forEach(key => {
39                  validateString(key);
40                  validateString(String(request.query[key]));
41              });
42              next();
43          } catch (e) {
44              console.log(`query parameters validation error: ${e}`);
45              return response.sendStatus(400);
46          }
47      }
48  }
49
50  module.exports = {
51      bodyValidation,
52      queryParamValidation,
53  };
```

pets.js

```
const { PETS_ID } = require('../config');

class Pet {
    constructor(request) {
        const { id, clientId } = request.params;
        const { name, age, cuteness, type } = request.body ? request.body : {};
        Object.assign(this, { id, clientId, name, age, cuteness, type });
        this.typeId = PETS_ID;
    }

    get item() {
        const { id, clientId, name, age, cuteness, type } = this;
        return { id, clientId, name, age, cuteness, type };
    }

    get ppId() {
        return `${this.typeId}#${this.clientId}`;
    }

    get psId() {
        return this.id;
    }
}

module.exports = {
    Pet,
};
```

wildAnimals.js

```
const { WILD_ANIMALS_ID, COUNTER_ID } = require('../config');

class WildAnimal {
    constructor(request) {
        const { id, type } = request.params;
        const { age } = request.body ? request.body : {};
        Object.assign(this, { id, age, type });
        this.updateCounter = true;
        this.getCounter = true;
        this.typeId = WILD_ANIMALS_ID;
    }

    get item() {
```

```
        const { id, age, type } = this;
        return { id, age, type };
    }

    get ppId() {
        return `${this.typeId}#${this.type}`;
    }

    get psId() {
        return `${this.id}`;
    }

    get ppCountId() {
        return `${this.typeId}#${COUNTER_ID}`;
    }

    get psCountId() {
        return this.type;
    }
}

module.exports = {
    WildAnimal,
};
```

`items.js` serves as a facade for doing GET and POST calls, abstracting the type of animal.

items.js

```
const ddb = require('./dynamoDb');

function returnServerError(e, response) {
    console.error(e);
    return response.status(500).send({});
}

async function getItem(request, response, animal) {
    await ddb.getItem(animal)
        .then(data => {
            if (data && data.item) {
                return response.status(200).send(data.item);
            }
            return response.status(404).send({});
```

```
15          })
16          .catch(e => returnServerError(e, response));
17  }
18
19  async function postItem(request, response, animal) {
20      await ddb.putItem(animal)
21          .then(() => response.status(201))
22          .catch(e => returnServerError(e, response));
23  }
24
25  function callFor(AnimalClass) {
26      return {
27          get(request, response) {
28              return getItem(request, response, new AnimalClass(request));
29          },
30          post(request, response) {
31              return postItem(request, response, new AnimalClass(request));
32          },
33      };
34  }
35
36  module.exports = {
37      callFor,
38  };
```

Finally, `dynamoDb.js` is the place to be for anything related to the database. It contains DB setup, parameters, calls, transformations of returned values, etc.

dynamoDb.js

```
1   const AWS = require('aws-sdk');
2   const { DDB_TABLE } = require('../config');
3
4   const docClient = new AWS.DynamoDB.DocumentClient({region: 'eu-west-1'});
5
6   async function dynamoGet(params) {
7       return docClient.get(params).promise();
8   }
9
10  async function dynamoPut(params) {
11      return docClient.put(params).promise();
12  }
13
```

```
14 async function dynamoUpdate(params) {
15     return docClient.update(params).promise();
16 }
17
18 function initCounter({ ppCountId, psCountId }) {
19     const dynamoDbParams = {
20         TableName: DDB_TABLE,
21         Item: {
22             ppId: ppCountId,
23             psId: psCountId,
24             itemCount: 0,
25         },
26         ConditionExpression: 'attribute_not_exists(itemCount)',
27     };
28     return dynamoPut(dynamoDbParams).catch(() => {
29         // error if the counter already exists
30     });
31 }
32
33 async function updateCounter(animal) {
34     await initCounter(animal);
35     const params = {
36         TableName: DDB_TABLE,
37         Key: {
38             ppId: animal.ppCountId,
39             psId: animal.psCountId,
40         },
41         UpdateExpression: 'Set itemCount = itemCount + :incr',
42         ExpressionAttributeValues: {
43             ':incr': 1,
44         },
45         ReturnValues: 'UPDATED_NEW',
46     };
47     return dynamoUpdate(params);
48 }
49
50 function getCounter(animal) {
51     const params = {
52         TableName: DDB_TABLE,
53         Key: {
54             ppId: animal.ppCountId,
55             psId: animal.psCountId,
56         },
```

```
    };
    return dynamoGet(params)
        .then(data => {
            if (data.Item) {
                return data.Item.itemCount;
            }
            return 0;
        });
}

function get(animal) {
    const params = {
        TableName: DDB_TABLE,
        Key: {
            ppId: animal.ppId,
            psId: animal.psId,
        },
    };
    return dynamoGet(params)
        .then(data => {
            if (data.Item) return { item: data.Item };
        });
}

async function getWithCounter(animal) {
    return Promise.all([
        get(animal),
        getCounter(animal),
    ])
        .then(data => {
            if (data[0]) {
                return {
                    item: {
                        ...data[0].item,
                        count: data[1],
                    },
                };
            }
        });
}

function put(animal) {
    const { ppId, psId, item } = animal;
```

```
    Object.assign(item, { ppId, psId });
    const dynamoDbParams = {
        TableName: DDB_TABLE,
        Item: item,
    };
    return dynamoPut(dynamoDbParams);
}

function putWithCounter(animal) {
    return put(animal)
        .then(() => updateCounter(animal));
}

function getItem(animal) {
    if (animal.getCounter) return getWithCounter(animal);
    return get(animal);
}

function putItem(animal) {
    if (animal.updateCounter) return putWithCounter(animal);
    return put(animal);
}

module.exports = {
    getItem,
    putItem,
};
```

Evaluation

The project has some qualities:

- Some thought has gone into organising the code and the file names give us a pretty good idea of what is contained within, even if “helpers” and “lib” could be more helpful.
- There are few dependencies, resulting in good performance.
- It is quite easy to add a new type of animal as long as no new special behavior is required. Storing information with or without a counter only requires the creation of a new class and routes.
- Right now, the code is not overly complicated (but this is also due to it being a mere example).

That being said, what about the downsides? What are the issues with this project specifically and what does that tell us about other projects written in a similar style?

First, the **passing of a 'request', 'response' and 'next' parameter**, something express seems to have made popular in the Node ecosystem. For example:

helpers.js

```
function queryParamValidation(request, response, next) {
    if (!request.query) {
        next();
    } else {
        try {
            // validate
            next();
        } catch (e) {
            return response.sendStatus(400);
        }
    }
}
```

Despite its popularity, this approach comes with a lot of downsides. Most important among those: a lot of code becomes 'aware' of what framework we are using. So if we decide to switch to a framework that does not inject parameters, we have to change a large part of our code base. Due to the dynamic nature of Javascript, changing to a similar-but-not-quite-the-same framework presents additional difficulties. Small differences might cause subtle bugs that slip by our tests and manual verifications. Especially because we will probably be mocking those three parameters in most tests.

Speaking of tests, if the framework 'infects' more of our code, we have to do more mocking, making our test code more complex and less predictable. A simple function that checks whether an incoming request contains an id should produce a return value that we can verify. With an express-like framework, we have to check the response parameter instead.

In the category of more general issues, the options provided by the framework encourage us to disrupt the flow of the application. We might stop and return an HTTP response wherever the response parameter is passed by calling `response.send()`. This means you either need detailed documentation or the code itself to understand the behavior of a specific function. It might pass on the framework parameters to the next function. Maybe it returns a response in some but not all cases. Thanks to the framework appearing everywhere, you need to have the entire application in your head to understand even a small part of it. To make matters even more difficult, the `next` parameter can be used to stop executing and continue to the… next function.

Finally, mutation of parameters is encouraged. You want some data to end up in functions downstream? Just add it to the request parameter! Convenient but dangerous for several reasons. First, people unfamiliar with your application will suddenly see unknown properties appearing in the request parameter. Second, we might accidentally destroy or change data (like the request body) which is needed further downstream.

Meanwhile the **classes** that are created for each type of animal (so in our case pets and wild animals) limit reuse. Observe how the class is injected into functions for getting information, for example from `getItem` in `items.js` to the underlying `getItem` and on to `getWithCounter` or `get`. Because of this, a majority of our functions need us to pass in an animal parameter, which itself *requires the incoming request parameter for its constructor*. Yet again, we are regrettably tying our code to the framework, as well as to our animal class.

This approach limits reuse both directly and indirectly. By the former I mean we can only use functions where we can build or fake the right animal object. The indirect issue, layering, is related. The underlying layer of, say, data retrieval should be usable by code without knowledge of higher concerns like HTTP requests (the request and response parameters). With the current setup, it is very difficult to 'hook in' to underlying layers without running into a bunch of bugs if you do not have the request parameter. Trying to retrieve animal information coming from a queue or topic instead of the gateway? Either duplicate code and make it work without an animal class, or fake that object, add a lot of tests and hope they cover all possible scenarios. And with the additional parameter, testing has again become more tiresome.

Now onto **dynamoDb.js**. Its main issue is doing too many things. About a third of all code is located in this file and it is three times bigger than most other files. What are its responsibilities? A whole bunch of things. It sets up the client for talking to DynamoDB, it calls AWS SDK methods, creates parameters for calls and applies some (but not all) 'transformations' to the data that is returned. As the project becomes more complex, it will grow uncontrollably. Want to add another animal with some additional behavior? Add a new file similar in size to `pets.js` plus a few lines for routing and `items.js`. Everything else will happen in `dynamoDB.js`. Want to add a delete? Write a few lines of routing; all the other code will go into our favorite file!

So on a macro level the file contains too much and does too much. And the same goes for the functions within. So if our application starts expanding, these functions will grow exponentially and will become harder to read, harder to reuse (because they become more specific) and harder to test, requiring you to mock away the whole AWS setup.

Error handling is not ideal and the reader may have already guessed the reasons. For one, it is spread out across the application. Sure you have a `helpers.js` which does some checking of incoming parameters and body. But take a look at `items.js` and you will see it handling some additional scenarios:

items.js

```
async function getItem(request, response, animal) {
    await ddb.getItem(animal)
        .then(data => {
            if (data && data.item) {
                return response.status(200).send(data.item);
            }
            return response.status(404).send({});
        })
        .catch(e => returnServerError(e, response));
}
```

There is also a mix of ways in which errors are handled. In one part of the application we throw errors when something goes wrong. In another we catch errors and return responses using the passed-in parameter. This approach complicates the flow of the application. When I throw an error, other functions may or may not capture it. And I have to go through the code to find out what actually happens.

The configuration file is problematic because it contains both constants and environment variables. Why is this an issue? First it makes testing slightly harder because we have to make sure the environment variables are set. Second, we have once again made all our functions 'aware' of where the configuration is coming from. That's none of their business damnit! Third, most functions require only constants but no (changeable) environment variables. Yet they receive both instead of being limited to what they actually need.

These remarks may sound like nitpicking, but some months after first pointing this out as a potential flaw in a real application (but not doing anything about it-whoops!), it actually caused a serious bug. Two Lambdas were using the same source code, including a configuration file. A feature was written for Lambda A, which required an additional environment variable. That variable was added to the infrastructure and to the configuration. Nothing was changed for Lambda B because it did not require the new variable. But it started crashing when it tried to load the updated configuration because it did not have a value for the new variable and there was a check to make sure a value was present! These crashes would have been avoided if Lambda B had loaded only the constants and environment variables it required.

Finally, some additional **minor issues**: business logic is spread out across the application instead of being the core. But this is a simple CRUD application with only limited specific business-related code, so maybe that's acceptable. `forEach` is used in several locations. Which is okay but does point to mutation, which can lead to more buggy applications. Furthermore, the classes are mostly used as data containers. Which is not bad per se. But why use full-fledged classes if you only want to store some data?

Improvements overview

My ideas around how this application could be improved fall into two categories: general advice that should be applicable to any type of code and advice related to Functional Programming.

First, regarding the **framework and its parameters**: pass such parameters to as few functions as possible. You will want to keep everything related to receiving and responding as close as possible to the 'surface' of the application. It should be in the outer layer of a hexagonal architecture. In a layered architecture, these kinds of concerns should only appear in the controller but never in the underlying service or database layers. This will make a large part of your application easier to understand and test.

FP helps with its focus on pure functions and immutability. When writing such functions, you try to avoid things like request/response parameters altogether and at a minimum a focus on purity will encourage you to only use them in a limited number of places. Immutability will avoid bugs and complexity caused by mutating the incoming request and response. So by default, FP encourages you to follow the advice of not infecting your entire code with these framework concerns.

As far as the **classes** are concerned, the general advice is similar. Make sure the specifics of your framework-and the classes that use it-do not appear throughout your code. Limit impact. Code that can handle generic data will be more reusable. Another bit of general advice would be to use types to describe the needs of functions. Our functions will be reusable because they are not tied to a specific class, but they do tell us what they need via their type. This gives us some additional assurances when 'hooking into' a part of our code, with the compiler warning us if this won't work.

FP also encourages us to make our functions as generic as possible, capable of working with a wide variety of data structures. Functions like `map` and `filter` care little about your underlying class or object. They just work, regardless of the type of data they are working with. For such functions encapsulation (which is very important in the world of OOP) is actually a downside. Encapsulate a generic function inside a class so it can work with its private variables and you have to write it every time you create a new class.

The **dynamoDB file** should be split up and its functions simplified. From Object Oriented Programming we can recall the SOLID principle of 'Single Responsibility' to guide us. And from FP we have learned to write pure functions focussed on a single task which can be composed to perform more complex behavior. This should improve readability and testability.

As far as **error handling** is concerned, we already know how to use monads to handle failure scenarios. That way we avoid breaking our flow while making our types more informative, with signatures showing what kind of unhappy paths we might face. Concentrating our error handling might also be beneficial, and this is something we have done in our projects as well.

The general advice for changing the **configuration file** is probably self-evident: split up the real constants and the injected environment variables. As we have seen in earlier chapters, we could use an IO monad to retrieve the configuration. If we also used a type for the injected config, as we did before, this will warn us at compile time that an environment variable was missing.

Finally, in a more complex application (beyond simple CRUD) we would use DDD and perhaps a hexagonal architecture to make sure our business logic was clearly separated from other concerns. FP encourages us once again to use immutability to avoid issues regarding mutation.

Our new code setup

Let us turn theory into practice and rewrite the application in a functional style, using techniques from earlier chapters to improve aspects like the readability and testability of the application. Our goal will be to start using composition of pure functions as much as possible, whilst avoiding mutation. The result will undoubtedly not be perfect(ly functional). But in spirit it will be far removed from the first version. And just to be clear, we are rewriting the application, not refactoring, because that would or should be piece-by-piece, keeping the existing application up and running. Refactoring is the more useful approach because you are taking an existing product and making it better while keeping it running. But rewriting is simpler to explain and allows for easier side-by-side comparison.

We will start by showing our new and improved Pet Service before explaining what we did. The `setup.sh` is unchanged except for the name of the stack. The infrastructure is basically the same, though the location of the Lambda zip and handler have changed.

setup.sh

```
#!/bin/bash

set -eo pipefail

STACKNAME="pets-example-new"

// everthing remains the same
```

infrastructure.yaml

```
AWSTemplateFormatVersion: 2010-09-09
Transform: AWS::Serverless-2016-10-31

// Parameters are the same

Resources:
  DefaultLambda:
    Type: AWS::Serverless::Function
    DependsOn: LambdaLogGroup
    Properties:
      FunctionName: !Ref AWS::StackName
```

```
12        CodeUri: dist/pets.zip ### this has changed
13        Runtime:  nodejs14.x
14        Timeout: 10
15        MemorySize: 1024
16        Handler: main.handler ### this has changed
17
18  // everything else still the same
```

Because we switched to Typescript, we added a `tsconfig.json` and a webpack configuration.

tsconfig.json

```
1  {
2    "compilerOptions": {
3      "lib": [
4        "es2017",
5        "esnext"
6      ],
7      "esModuleInterop": true,
8      "moduleResolution": "node",
9      "noUnusedLocals": true,
10     "noUnusedParameters": true,
11     "sourceMap": true,
12     "target": "es2017",
13     "outDir": "dist",
14     "noImplicitThis": true,
15     "noImplicitAny": true,
16     "strictBindCallApply": true
17   },
18   "exclude": [
19     "node_modules",
20     "dist"
21   ]
22 }
```

webpack.config.js

```
const path = require("path");

module.exports = {
    mode: 'production',
    resolve: {
        extensions: [".js", ".jsx", ".json", ".ts", ".tsx"],
    },
    output: {
        libraryTarget: "commonjs",
        path: path.join(__dirname, "dist"),
        filename: "main.js",
    },
    target: "node",
    externals: ["aws-sdk"],
    module: {
        rules: [{ test: /\.tsx?$/, loader: "ts-loader" }],
    },
};
```

The change to Typescript has also introduced some changes in our `package.json`, with new dependencies added and the old one for lambda-api taken out. These dependencies and scripts should be familiar by now.

package.json

```
{
  "name": "pet-service-new",
  "version": "1.0.0",
  "description": "New version of the pet service",
  "scripts": {
    "clean": "rm -rf node_modules && rm -rf dist",
    "webpack": "webpack",
    "zip": "cd dist && zip -r pets.zip *",
    "build": "npm i && webpack && npm run zip"
  },
  "dependencies": {
    "fp-ts": "2.8.3",
    "fp-ts-std": "^0.8.0"
  },
  "devDependencies": {
    "@types/aws-lambda": "^8.10.72",
    "aws-sdk": "^2.764.0",
    "ts-loader": "^5.3.3",
```

```
    "typescript": "^4.1.2",
    "webpack": "^5.3.0",
    "webpack-cli": "^4.1.0"
  },
  "author": "Sam Van Overmeire",
  "license": "ISC"
}
```

Inside the src folder, we have `index.ts`, `routing.ts` and `constants.ts`, containing some basic functions.

The index file is once again our Lambda entry point. It does very little except passing on the incoming event to the routing file. The advantage of doing very little in `index.ts` is that it makes our code easier to test. For example, we could use the file for the setup of dependencies that take a long while to load, and if we do this outside the handler function, they only get loaded once per Lambda container and we do not have to pay for the time needed to initialize them. This is a well-known Lambda optimization. But it does make the index file harder to test (even without calling the handler, just importing the file will run this setup). Hence the simple fix of moving our logic to other files. Plus, if we pass on these dependencies as parameters, replacing them with mocks and unit testing become very easy.

index.ts

```
import {APIGatewayProxyEventV2} from "aws-lambda/trigger/api-gateway-proxy";
import {route} from "./routing";

export const handler = async (event: APIGatewayProxyEventV2) => {
    console.log(JSON.stringify(event));
    return route(event);
};
```

constants.ts

```
export const GET_METHOD = 'GET';
export const POST_METHOD = 'POST';

export const WILD_ANIMALS_PREFIX = 'W';
export const COUNTER_PREFIX = 'C';
export const PETS_ID_PREFIX = 'P';
```

We avoid libraries that wrap information in request and response parameters for our routing. Instead, we write some simple code ourselves, good enough for demonstrative purposes and simple projects. The biggest issue is how it throws an error (side effect!) if it cannot find the right route. Though you

may have an idea how to fix that, we will keep it as is and come back to this issue later on in this chapter.

routing.ts

```
import {APIGatewayProxyEventV2} from "aws-lambda/trigger/api-gateway-proxy";
import {GET_METHOD, POST_METHOD} from "./constants";
import {getPet, postPet} from "./pets/entrypoint";
import {getWildAnimal, postWildAnimal} from "./wildAnimals/entrypoint";

// simplified by using query params instead of path
export const route = async (event: APIGatewayProxyEventV2) => {
    const path = event.requestContext?.http?.path;
    const method = event.requestContext?.http?.method;

    if (path === '/pets') {
        if (method === GET_METHOD) {
            return getPet(event);
        } else if(method === POST_METHOD) {
            return postPet(event);
        }
    } else if(path === '/wild') {
        if (method === GET_METHOD) {
            return getWildAnimal(event);
        } else if(method === POST_METHOD) {
            return postWildAnimal(event);
        }
    }
    throw new Error(`Invalid route for ${JSON.stringify(event)}`);
};
```

Unto the `src` folder. The `common` folder helps with code reuse. Anything that seems generic enough that it can be shared between creatures is placed there. It contains five files. First, a `types.ts` file for all our types, across folders. As usual we add `readonly` to properties. Note the ADT/Sum type.

types.ts

```
export type RetrievePetRequest = {
    readonly id: string,
    readonly clientId: string,
};

export type AddPetRequest = {
    readonly id: string,
    readonly clientId: string,
    readonly name: string,
    readonly age: number,
    readonly cuteness: string,
    readonly type: string,
};

export type RetrieveWildAnimalRequest = {
    readonly id: string,
    readonly type: string,
};

export type AddWildAnimalRequest = {
    readonly id: string,
    readonly type: string,
    readonly age: number,
};

/* errors and responses */

export type Response = {
    readonly statusCode: number,
    readonly body?: string,
}

export type ClientError = {
    readonly type: 'ClientError',
    readonly message: string,
}

export type ServerError = {
    readonly type: 'ServerError',
    readonly message: string,
}

export type NotFoundError = {
```

```
    readonly type: 'NotFoundError',
}

export type Errors = ClientError | ServerError | NotFoundError;
```

The `Errors` types are created by functions inside `errors.ts`:

errors.ts

```
import {ClientError, NotFoundError, ServerError} from "./types";

export const clientError = (message: string): ClientError => {
    return {
        type: "ClientError",
        message,
    }
};

export const serverError = (message: string): ServerError => {
    return {
        type: "ServerError",
        message,
    }
};

export const notFound = (): NotFoundError => {
    return {
        type: "NotFoundError",
    };
};
```

`gateway.ts` contains some generic functions for communicating with DynamoDB:

gateway.ts

```
import {DynamoDB} from "aws-sdk";
import {DocumentClient} from "aws-sdk/lib/dynamodb/document_client";
import GetItemInput = DocumentClient.GetItemInput;
import PutItemInput = DocumentClient.PutItemInput;
import GetItemOutput = DocumentClient.GetItemOutput;
import UpdateItemInput = DocumentClient.UpdateItemInput;
import * as A from "fp-ts/Array";
import * as TE from "fp-ts/TaskEither";
import {serverError} from "./errors";
import {Errors} from "./types";

const client = new DynamoDB.DocumentClient({region: 'eu-west-1'});

export const get = (params: GetItemInput) => client.get(params).promise();

export const post = (params: PutItemInput) => client.put(params).promise();

export const update = (params: UpdateItemInput) => client.update(params).promise();

export const getParallel = (inputs: GetItemInput[]): TE.TaskEither<Errors, GetItemOu\
tput[]> => {
    return A.array.sequence(TE.taskEither)(
        inputs.map(input => TE.tryCatch(() => get(input), serverError))
    );
};

export const getItem = (v: GetItemOutput) => v.Item;
```

`responses.ts` can build http responses to return from the Lambda:

responses.ts

```
import {Errors, Response} from "./types";

export const okResponse = (body: string): Response => {
    return {
        statusCode: 200,
        body,
    }
};

export const createdResponse = (): Response => {
    return {
```

```
12          statusCode: 201,
13      }
14  };
15
16  export const badRequestResponse = (message: string): Response => {
17      return {
18          statusCode: 400,
19          body: message,
20      }
21  };
22
23  export const notFoundResponse = (): Response => {
24      return {
25          statusCode: 404,
26      }
27  }
28
29  export const internalServerErrorResponse = (): Response => {
30      return {
31          statusCode: 500,
32          body: 'Internal Server Error',
33      };
34  }
35
36  export const handleError = (err: Errors) => {
37      switch (err.type) {
38          case 'ClientError': {
39              return badRequestResponse(err.message);
40          }
41          case 'NotFoundError': {
42              return notFoundResponse();
43          }
44          case 'ServerError': {
45              console.warn(err.message);
46              return internalServerErrorResponse();
47          }
48          default: {
49              const oops: never = err;
50              return oops;
51          }
52      }
53  }
```

And `validation.ts` contains a couple of validation helpers:

validation.ts

```
import {DocumentClient} from "aws-sdk/lib/dynamodb/document_client";
import GetItemOutput = DocumentClient.GetItemOutput;
import {APIGatewayProxyEventV2} from "aws-lambda/trigger/api-gateway-proxy";
import * as E from "fp-ts/Either";
import * as TE from "fp-ts/TaskEither";
import * as A from "fp-ts/Array";
import {Errors} from "./types";
import {clientError, notFound} from "./errors";

const inValid = (value: string) => !value.trim();

export const checkQueryString = (event: APIGatewayProxyEventV2): E.Either<Errors, AP\
IGatewayProxyEventV2> =>
    Object.entries(event.queryStringParameters).some(entry => inValid(entry[0]) || i\
nValid(entry[1])) ?
        E.left(clientError('Query string validation error')) : E.right(event);

export const checkBody = (parsedBody: E.Json) => {
    return Object.keys(parsedBody).some(inValid) ?
        E.left(clientError('Body contained empty key')) : E.right(parsedBody);
};

const checkEmpty = (item: GetItemOutput): E.Either<Errors, GetItemOutput> => item.It\
em ? E.right(item) : E.left(notFound());

export const checkEmptyTE = (res: GetItemOutput) => TE.fromEither(checkEmpty(res));

export const parallelCheckEmpty = (inputs: GetItemOutput[]): TE.TaskEither<Errors, G\
etItemOutput[]> => {
    return A.array.sequence(TE.taskEither)(
        inputs.map(input => checkEmptyTE(input))
    );
};
```

The other folders under `src` contain our pets and wild animals implementations. One thing to note is that the structure has changed. Both animal types now have their own folder (one is `pets` and the other `wildAnimals`) containing everything they need! This is based on DDD advice: we should group our code according to the domain instead of technical detail. The reason for this is that whenever you change something (fix a bug, add a feature), you only have one place (one folder) where code changes

are required. Currently each animal folder contains three files: `anticorruption.ts`, `database.ts` and `entrypoint.ts`. For now we will be focussing on validation, so we will look at the entrypoint and anticorruption files. We will show the database files when the time is ripe.

Validation

Let us start with the `entrypoint.ts` files. The name tells us that these are the entrypoints to all the code in this specific folder. (Naming is extremely hard and the names I picked are best effort.)

entrypoint.ts (pets)

```
import * as TE from 'fp-ts/TaskEither';
import * as T from 'fp-ts/Task'
import {APIGatewayProxyEventV2} from "aws-lambda/trigger/api-gateway-proxy";
import {pipe} from "fp-ts/function";
import {preparePetGet, preparePetPost} from "./anticorruption";
import {putInDatabase, retrieveFromDatabase} from "./database";
import {createdResponse, handleError, okResponse} from "../common/responses";
import {Response} from "../common/types";

export const getPet = async (event: APIGatewayProxyEventV2): Promise<Response> =>
    pipe(
        preparePetGet(event),
        TE.fromEither,
        TE.chain(retrieveFromDatabase),
        TE.map(JSON.stringify),
        TE.bimap(handleError, okResponse),
        TE.getOrElse(T.of)
    )();

export const postPet = async (event: APIGatewayProxyEventV2): Promise<Response> =>
    pipe(
        preparePetPost(event),
        TE.fromEither,
        TE.chain(putInDatabase),
        TE.bimap(handleError, createdResponse),
        TE.getOrElse(T.of)
    )();
```

entrypoint.ts (wild animals)

```
import {APIGatewayProxyEventV2} from "aws-lambda/trigger/api-gateway-proxy";
import {pipe} from "fp-ts/function";
import * as TE from "fp-ts/TaskEither";
import * as T from "fp-ts/Task";
import {prepareWildAnimalGet, prepareWildAnimalPost} from "./anticorruption";
import {createdResponse, handleError, okResponse} from "../common/responses";
import {putInDatabase, retrieveFromDatabase} from "./database";
import {Response} from "../common/types";

export const getWildAnimal = async (event: APIGatewayProxyEventV2): Promise<Response\
> =>
    pipe(
        prepareWildAnimalGet(event),
        TE.fromEither,
        TE.chain(retrieveFromDatabase),
        TE.map(JSON.stringify),
        TE.bimap(handleError, okResponse),
        TE.getOrElse(T.of)
    )();

export const postWildAnimal = async (event: APIGatewayProxyEventV2): Promise<Respons\
e> =>
    pipe(
        prepareWildAnimalPost(event),
        TE.fromEither,
        TE.chain(putInDatabase),
        TE.bimap(handleError, createdResponse),
        TE.getOrElse(T.of)
    )();
```

Now focus on the pets example. What we can see is that there are functions for GET and POST. If we wanted a DELETE, we would add a `delete` here that would be called by the 'outside' code (i.e. anything outside the pets folder). This is the first reason for having entrypoints: they limit the surface area of our pets module. Sadly, other parts of the application can still import from all the other files, as there is currently no way to force a more local export in Javascript/Typescript. So to enforce this, rely on agreement within the team or write some code to do this automatically.

In any case, this surface area limitation is especially useful for functional applications since a large number of functions combined with a large number of files and importing from anywhere can quickly become exponentially confusing. There are similar, well-known rules for structuring layered (controller-service-database) applications.

The second reason for having entrypoints is that we need a place to combine the various pieces of our application, composing our application from smaller functions. The above functions do just that:

they take the incoming event and send it through a pipeline, returning a result at the end.

We validate the incoming events as soon as they enter our folder using `preparePetGet` and `preparePetPost`. This functionality comes from our anticorruption file. Looking at the exported functions, we can see that it's turtles all the way down (alternatively, some reference to the movie Inception) because the prepare* functions are themselves composed of functions:

anticorruption.ts (pets)

```
import * as E from 'fp-ts/Either'
import {APIGatewayProxyEventV2} from "aws-lambda/trigger/api-gateway-proxy";
import {pipe} from "fp-ts/function";
import {clientError} from "../common/errors";
import {AddPetRequest, Errors, RetrievePetRequest} from "../common/types";
import {checkBody, checkQueryString} from "../common/validation";

const parseEvent = (event: APIGatewayProxyEventV2): RetrievePetRequest => ({
    id: event.queryStringParameters.id,
    clientId: event.queryStringParameters.clientId,
});

const parseEventFromBody = (parsedBody: E.Json): AddPetRequest => {
    const records = parsedBody as E.JsonRecord;

    return {
        id: records.id as string,
        clientId: records.clientId as string,
        name: records.name as string,
        age: records.age as number,
        cuteness: records.cuteness as string,
        type: records.type as string,
    };
};

export const preparePetGet = (event: APIGatewayProxyEventV2): E.Either<Errors, Retri\
evePetRequest> =>
    pipe(
        checkQueryString(event),
        E.map(parseEvent),
    );

export const preparePetPost = (event: APIGatewayProxyEventV2): E.Either<Errors, AddP\
etRequest> =>
    pipe(
```

```
        E.parseJSON(event.body, () => clientError('Invalid json body')),
        E.chain(checkBody),
        E.map(parseEventFromBody),
    );
```

anticorruption.ts (wild animals)

```
import {APIGatewayProxyEventV2} from "aws-lambda/trigger/api-gateway-proxy";
import * as E from "fp-ts/Either";
import {pipe} from "fp-ts/function";
import { AddWildAnimalRequest, Errors, RetrieveWildAnimalRequest } from "../common/t\
ypes";
import {checkBody, checkQueryString} from "../common/validation";
import {clientError} from "../common/errors";

const parseEvent = (event: APIGatewayProxyEventV2): RetrieveWildAnimalRequest => ({
    id: event.queryStringParameters.id,
    type: event.queryStringParameters.type,
});

const parseEventFromBody = (parsedBody: E.Json): AddWildAnimalRequest => {
    const records = parsedBody as E.JsonRecord;

    return {
        id: records.id as string,
        age: records.age as number,
        type: records.type as string,
    };
};

export const prepareWildAnimalGet = (event: APIGatewayProxyEventV2): E.Either<Errors\
, RetrieveWildAnimalRequest> =>
    pipe(
        checkQueryString(event),
        E.map(parseEvent),
    );

export const prepareWildAnimalPost = (event: APIGatewayProxyEventV2): E.Either<Error\
s, AddWildAnimalRequest> =>
    pipe(
        E.parseJSON(event.body, () => clientError('Invalid json body')),
        E.chain(checkBody),
```

```
        E.map(parseEventFromBody),
    );
```

The exposed functions have two tasks. First, they check the values contained inside the incoming event. For a GET we check the query string, for a POST the body. These functions are small, focused on one task and they output a value that depends solely on the input provided via parameters. In other words, these are pure functions, which makes them easy to compose. As we did in earlier chapters, we use the Either monad to compose our functionality without resorting to exceptions. For example `checkQueryString`, which checks for empty names or values in the query string parameters:

validation.ts (common)

```
const inValid = (value: string) => !value.trim();

export const checkQueryString = (event: APIGatewayProxyEventV2): E.Either<Errors, AP\
IGatewayProxyEventV2> =>
    Object.entries(event.queryStringParameters).some(entry => inValid(entry[0]) || i\
nValid(entry[1])) ?
        E.left(clientError('Query string validation error')) : E.right(event);
```

The monads are combined by using `chain` and `map`, which is what we see happening in our exposed functions like `preparePetPost`. We profit from our ability to easily combine pure functions in order to expand our application logic.

The second task of `anticorruption.ts` is to take the incoming payload and transform it. This is done by `parseEventFromBody`, which unlike earlier calls returns a normal value, which is why it is combined with the previous functions using `map`. We want to avoid contaminating our entire code base with concepts related to AWS and HTTP (`APIGatewayProxyEventV2`), so once we know the request is valid we take only what we need and use it to build a new object. The type that we return from `preparePetPost`, called `RetrievePetRequest`, only contains an id and client id, which is all we need to find pets.

So `anticorruption.ts` ends with us getting either an error message or an object that we can use further down the road. In the latter case, we can do a few transformations, call the database and return something to the user. We will concentrate on that scenario later on. Right now we want to continue exploring error handling.

You may remember from `types.ts` that our `Errors` type is an ADT, a Sum type. We have used a Sum type before as it allows us to have 'subtypes' with different fields, giving us a lot of flexibility. If we were forced to add a useless message field to `NotFoundError` only because another kind of error required it (a subclassing scenario), that might lead to ugly code and dangerous situations.

And yes, in the case of our simple application, we could have just gone for a string to indicate our error. But that 'simplicity' comes at a cost. First, it limits our options. What about that `NotFoundError` where we had no interesting fields whatsoever? Do we leave the string as null or empty and assume

we have a `NotFoundError` if that is the case? Fill in "NotFound" and take some specific action? Second, our type would not offer a lot of additional information to the compiler. A string can take a lot of values and all would be accepted as valid errors values. Meanwhile our `Errors` type is currently limited to just three values! Put anything else in there and the compiler knows something is wrong.

By using a Sum Type we can easily indicate the type of error and are free to pass along any information we find useful. Perhaps the path of the request would be useful for debugging a caller issue? Sure, just add a path property to ClientError! And as we have seen before, we can use pattern matching to handle all `Errors` in their own particular way:

responses.ts (common)

```
export const handleError = (err: Errors) => {
    switch (err.type) {
        case 'ClientError': {
            return badRequestResponse(err.message);
        }
        case 'NotFoundError': {
            return notFoundResponse();
        }
        case 'ServerError': {
            console.warn(err.message);
            return internalServerErrorResponse();
        }
        default: {
            const oops: never = err;
            return oops;
        }
    }
}
```

Using a switch, we return a proper response for each kind of error. A client error leads to a bad request (400) and we pass on the message we created earlier as additional information. For a not found, we send a 404 without a message. The extra conveniences rest with the compiler. Typescript is smart enough to know the exact 'subtype' we are dealing with once we enter a specific part of the switch, meaning it can warn us if a particular property does not exist.

We also do an exhaustiveness check by returning something of type `never`, which is impossible and forces Typescript to warn us. Meaning when we add a new type of error, our code will fail at compile time if we do not take action in `handleError`. Since this function is called by our entrypoint functions, by the end of those functions we either have a response to give back to the caller, or a proper error status code.

Dealing with the asynchronous calls

Let's take another look at the entrypoint of our pets folder. Here is some relevant code:

entrypoint.ts (pets)

```
export const getPet = async (event: APIGatewayProxyEventV2): Promise<Response> =>
    pipe(
        preparePetGet(event),
        TE.fromEither,
        TE.chain(retrieveFromDatabase),
        TE.map(JSON.stringify),
        TE.bimap(handleError, okResponse),
        TE.getOrElse(T.of)
    )();

export const postPet = async (event: APIGatewayProxyEventV2): Promise<Response> =>
    pipe(
        preparePetPost(event),
        TE.fromEither,
        TE.chain(putInDatabase),
        TE.bimap(handleError, createdResponse),
        TE.getOrElse(T.of)
    )();
```

We discussed the 'prepare' step where we validate the event we received and transform it into a domain object. We also see some functions related to the database (`retrieveFromDatabase` and `putInDatabase`). As these are asynchronous calls, we once again need to turn to that very useful monad transformer called TaskEither. As we saw in earlier chapters, we can use `fromEither` to lift our existing monad into a TaskEither, making our subsequent work with our new favorite monad easier.

Going further downstream to our database files, we see that it's composition and piping all the way down, since all the exported functions are composed of smaller functions:

database.ts (pets)

```
const buildGet = (request: RetrievePetRequest) => ({
    TableName: process.env.DATABASE_NAME,
    Key: {
        ppId: `${PETS_ID_PREFIX}#${request.clientId}`,
        psId: request.id,
    },
});

const buildPost = (request: AddPetRequest) => ({
    TableName: process.env.DATABASE_NAME,
    Item: {
        ppId: `${PETS_ID_PREFIX}#${request.clientId}`,
        psId: request.id,
        id: request.id,
        clientId: request.clientId,
        name: request.name,
        age: request.age,
        cuteness: request.cuteness,
        type: request.type
    }
});

export const retrieveFromDatabase = (request: RetrievePetRequest) =>
    pipe(
        buildGet(request),
        (params) => TE.tryCatch(() => get(params), serverError),
        TE.chain(checkEmptyTE),
        TE.map(getItem)
    );

export const putInDatabase = (request: AddPetRequest) =>
    pipe(
        buildPost(request),
        (params) => TE.tryCatch(() => post(params), serverError),
    );
```

The first step is building the right parameters for our call, for which we have some simple, pure functions. In the next step, we pass these params to the actual DynamoDB call inside the TaskEither constructor `tryCatch`. If something goes wrong, we will be returning a server error. After retrieving our item, we do some additional verification and mapping: we return a left with a `NotFoundError` if the returned value is empty. If we do find something, we put it in a right. As we saw before, there is some duplication and common logic and again, to me the cost in readability — and perhaps complexity — suggests to leave things 'as is' for now.

Now back to the entry point, where we use `bimap` to transform the left and right values that we received in previous steps. `handleError` was discussed previously, while in case of a success we either return a 200 http status (using `okResponse`) or a 201 (with `createdResponse`). Once this is done, we use `getOrElse` to unwrap our TaskEither and get the underlying Task. Since we already handled errors in the previous step, here we only need to tell the code to wrap any errors in a Task so that, whichever path we take, we get back the same thing. And since a Task is just a function returning a Promise, we end our function by invoking it.

Remember how I previously said that throwing an error in our routing was far from ideal? You probably know how to fix that, but just in case you memory is failing you: what we could do is return the TaskEither instead of 'unwrapping' it in our entry point. In which case the router starts with a (Task)Either and checks whether the path is valid, producing a client error if this is not the case. Next, we chain the proper entrypoint function (which now returns a TaskEither), use `getOrElse` and do a function call.

Alternatively, you could check out the small example library[30] I wrote, which features a very similar but not identical approach. Its main function returns a TaskEither, making *you* the one to decide when to actually call the underlying Promise. Keeping *my* stuff nice and pure until you, the caller, mess everything up by actually running the code.

Now let us take a peek at the wild animals implementation. Since the validation and entrypoint code are almost identical, we can focus on the database part:

database.ts (wild animals)

```
1  const buildGet = (request: RetrieveWildAnimalRequest) => {
2      return {
3          TableName: process.env.DATABASE_NAME,
4          Key: {
5              ppId: `${WILD_ANIMALS_PREFIX}#${request.type}`,
6              psId: request.id,
7          },
8      };
9  };
10
11 const buildCounterGet = (request: RetrieveWildAnimalRequest) => {
12     // similar, build some params
13 };
14
15 const buildPost = (request: AddWildAnimalRequest) => ({
16     // similar, build some params
17 });
18
19 const buildCounterInit = (request: AddWildAnimalRequest) => ({
```

[30] https://www.npmjs.com/package/fp-ts-lambda-routing

```
20      // similar, build some params, with an exists check
21  });
22
23  const buildCounterUpdate = (request: AddWildAnimalRequest): UpdateItemInput => ({
24      TableName: process.env.DATABASE_NAME,
25      Key: {
26          ppId: `${WILD_ANIMALS_PREFIX}#${COUNTER_PREFIX}`,
27          psId: request.type,
28      },
29      UpdateExpression: 'Set itemCount = itemCount + :incr',
30      ExpressionAttributeValues: {
31          ':incr': 1,
32      },
33      ReturnValues: 'UPDATED_NEW',
34  });
35
36  export const retrieveFromDatabase = (request: RetrieveWildAnimalRequest) => {
37      return pipe(
38          [buildGet(request), buildCounterGet(request)],
39          requests => getParallel(requests),
40          TE.chain(parallelCheckEmpty),
41          TE.map(results => results.map(getItem))
42      );
43  };
44
45  export const putInDatabase = (request: AddWildAnimalRequest) => {
46      return pipe(
47          buildCounterInit(request),
48          (params) => TE.tryCatch(() => post(params), serverError),
49          TE.fold(() => TE.right(request), () => TE.right(request)),
50          TE.chainFirst((params) => TE.tryCatch(() => post(buildPost(params)), serverE\
51  rror)),
52          TE.chainFirst((params) => TE.tryCatch(() => update(buildCounterUpdate(params\
53  )), serverError))
54      );
55  };
```

Because we have to retrieve a counter for the wild animals, there is a bit more code here for creating the parameters for the database calls. (Also, there is again some duplication here which we could fix, as well as passing in of environment variables.)

Let us look at GETting a wild animal with its counter. First we build the parameters for retrieving the animal as well as the right counter. Because there is no dependency between these calls, we do

them in parallel. Note that our `getParallel` method is a) very generic and was thus placed in the common section and b) builds on existing code (the generic get):

Because we get back multiple values now, we also have multiple empty checks to perform and we again use the already familiar fp-ts `sequence` and reuse our code for checking a single output. Finally, we do an Array `map` within the TaskEither `map`.

The POST is a little bit different, because the original code, which we are rewriting, first initializes the counter then posts the animal and only then updates the counter. So we should do something similar, but with monads. If you memorized the code of the original application (you weirdo), you also might remember that we catch whatever error comes back from the counter initialization because such an error probably just indicates that the counter already exists. There are several possible approaches. We could imitate the original code, wrapping our post inside the TaskEither in a try/catch. But that would result in ugly and misleading code: we are working with a TaskEither but in fact no error is ever thrown!

A viable alternative would be to first use a Task, catching all errors, and lifting that Task to a TaskEither, as we did before with Either. That way, our types are more accurate. Here, we chose a third approach: keep the init counter call inside a TaskEither (which is honest because the call might throw errors) but ignore failures in `putInDatabase` by using `fold` to always return the original request. This makes catching the error a more explicit part of our flow at the cost of a perhaps silly fold step. The next two steps use `chainFirst` for our calls to save the animal and update the counter, which is similar to a chain except it will pass on the original incoming parameter, i.e. our request. We do this because our update counter has no need of the post value, but does require the original request to create a key.

Advantages of our functional approach

Let's think about the advantages of this approach over our earlier 'naive' version. But before doing that, we can discuss some possible improvements as an exercise for the reader. For example, our rework contains no **tests**. But because we focussed on creating pure functions and eliminating side effects, adding those would be easy.

Also, there is some **duplication**. The higher level functions in `entrypoint.ts` and `anticorruption.ts` are very similar for pets and wild animals. In a larger project, we might create a function that accepts other functions for more specific behavior like creating the right domain object. This is similar to the Template Pattern from OOP, but simpler thanks to the versatility of higher-order functions. However, in a small project like this, such optimisations might be premature and work to our disadvantage. Maybe the access or validation patterns of wild animals and pets are currently similar by accident and will grow apart in the future? If that is the case and we have already moved this seemingly generic functionality to a common section, this function might become a complicated mess with a lot of conditionals.

Another issue is the **simplistic routing**, but we have already discussed how to fix that. Finally, the **injection of an environment variable** for the table name is especially glaring since we pointed

out that this was a downside in the original project! In a real application, we would retrieve these variables using an IO monad, injecting them into functions as parameters. If we had a lot of injection points, we would turn to the Reader monad.

Now the advantages. Our code is **organised by domain** which can make navigation easier. New features will for a large part be contained within a single folder. Want to add a new pet attribute like health? In the current setup, you only need to check out the pets folder and `types.ts`. (You might even refactor the application by putting specific types in the right folder.) Meanwhile the common folder has utility functions that will hopefully be generic enough that you do not have to explore the actual implementations. `get` retrieves an item, `getParallel` retrieves items in parallel. You have all the information you need in the name and type.

You may also have noticed that the basic calls to the database are in the common folder, in a file called 'gateway'. I did not come up with this name myself. I stole it since it seemed like a nice way to indicate a focus on communicating with the 'outside world'. Apart from reuse, keeping these calls separate from our domain logic in the other folders also improves testability, excepting those basic calls that are little more than wrappers around the AWS SDK - which is probably already tested pretty well by AWS itself… By isolating the actual calls to the outside world in a separate file, changing our database should also become easier. Do note that if this was our intent, we should have created our own parameters instead of littering our database files with AWS types like `GetItemInput`!

We have done our best to **limit our code surface**, creating only a few entry points to our pets and wild animals. Functions exported by other files in those folders should not be used anywhere outside of the folder. Once we pass beyond our entry points, 'hooking into' functions is now easier. Want to retrieve a pet? Just pass in any object containing an id and client id! In a larger application with a more meaningful domain, a large majority of functions would have easier, or at least sensible, types to work with. Here, with our limited domain logic, most have to do with receiving, validating and transforming the incoming event.

Our code consists of **short**, **pure functions** that are easy to test because they only depend on the given parameters. This makes covering all possible paths through the code easier, as does the limited amount of conditional branching. So with little effort we should be able to get high coverage. And because larger functionality is built by composing smaller functions, they too are easy to test and understand. Compare this to our earlier situation where we passed along parameters that could be mutated and had several side effects. In a real application, we would probably have more composition without monads (using fp-ts `flow`) and more higher-order functions, but the principles remain the same.

This focus on composition of small, focused functions stands in shrill contrast with the (relatively) large, ill-defined functions in `dynamoDb.js`. As an example of the former, see `getItem`, an extremely simple transformation that is reusable whenever we retrieve an item from the database. This helps keep complexity at bay, though we should take extra care in organising the code base, thinking long and hard about naming, folder structure, exports, etc.

Part of our application has now become **agnostic about the way we communicate** with the outside world. When an event enters our application, we change it into something our domain/code can

work with. And at the end, we transform our result into something appropriate for REST. This is obviously inspired by hexagonal architecture. We do now have a functional framework appearing in many layers of our code instead. In a language like Javascript, where we lack some essential components for writing functional code, this is unfortunately unavoidable. Perhaps one day the language itself will give us a range of monads to work with.

We are no longer throwing exceptions that might get caught anywhere (or nowhere). Instead our **flow has become straight and simple**. To replace exceptions we turned to the Either monad which forced us to handle the issue of invalid values, since our values are now 'stuck' inside the Either box. At some point we have to get them out!

Handling of errors is now done in one place and Typescript will warn us if we forget to handle a type of error. Speaking of **type checking**, ours is easygoing with decent type inference, yet still delivers value in making our application less error prone. And by setting every property in `types.ts` to `readonly`, Typescript makes sure no objects are mutated.

Performance, something the original application was good at, was never our primary concern. Still, webpack's tree-shaking means we get one small Javascript file instead of the relatively large original application zip (4 KB versus 170+ when unpacked). Though I have not done any proper comparisons, the few data points I do have point to slightly better average runtime and a very good minimum, just 5 ms. And since we now use a lot of pure functions, memoization is an easy way to speed things up even further when needed, with little impact on code complexity!

Summary

There are a lot of paradigms to write code in. OOP is the dominant one for most of the IT industry and FP is the focus of this book. But in languages like Javascript and Python, hybrid approaches are an option as well. Ideally, they offer a way to combine the strengths of OOP and FP. But here we looked at a less successful example of combining both approaches, exploring several issues that might haunt the project when it grows or changes. Next, we offered both a theoretical and practical overview of ways we could improve the application, at the same time making it more functional using what we learned in previous chapters.

We can now turn to the final chapter which briefly introduces some additional FP techniques.

Advanced FP techniques

The previous chapters explained many of the basic techniques and tools of functional programming and also explored the many ways we can use the power of monads. What you learned in those chapters is sufficient for writing a large variety of programs, but programmers have developed a few more advanced techniques that might be useful as well. Whether you need these will depend on the requirements and complexity of your projects. In the world of microservices or serverless, there is a good chance any individual application (like a single Lambda) contains code ranging from hundreds to thousands of lines length. If you work in that kind of an environment (I hope you do, I find it's a blast) hiding implementation, folder organisation and good names are often sufficient for keeping your project understandable & usable.

So we should proceed cautiously. It has often been said that the book on Design Patterns by the Gang of Four brought in its wake not just better design, as the authors hoped, but also a lot of useless complexity. Because some developers figured they just had to use design patterns wherever they could, instead of when it was appropriate. Similarly, **tagless finals**, **free monads** and even **DSLs** help navigate certain specific issues or complexity *and* are the cause of additional complexity at the same time. A final warning: I have only explored tagless finals and free monads in my spare time, not in any work-related projects. But I feel a brief overview definitely belongs in a book dedicated to FP.

Domain Specific Languages

Most of our previous work was focused on a small set of principles for building applications. They will probably sound familiar by now: create *small* and preferably *pure functions*, combine them with *composition* or *monads* to form larger functionality. Repeat this process until you have an application.

When building our applications, we also paid some attention to **modularisation**. Since we do not have classes to help us with information hiding, we use modules, exposing only the functions that are really needed by other parts of the application. By minimizing the *surface area* of our modules our code becomes more understandable to outsiders. Compare this to an application which exposes every possible function. While that would still be better than putting all our code in just one file (at least the names offer clues on where to find a relevant piece of code), it's still pretty bad. A developer looking at file A has to keep functions B, C, D… from files 1, 2, 3… (i.e. a lot of moving pieces) in mind when looking at your code. If those additional functions have a certain goal in common (say, handling authentication), it is better to place them all together and expose them via a minimal API.

As we said earlier, this can suffice for smaller applications, especially those where no clear domain has yet been discovered/found. For larger applications, the dominant technique of handling

complexity & domain modelling in the world of FP is creating **Domain Specific Languages**, or **DSLs**. In case you have not heard of DSLs, the title gives most of it away. You develop a language specifically suited for the domain you are working in, either by using an existing language and molding it to your liking, or by developing an entirely new one! The former is obviously the easier way of doing it, while the latter offers more options and flexibility.

Your DSL will probably be based on the **ubiquitous language** that developers and domain experts developed together based on their shared understanding of the domain. And because it mimics this shared language, a DSL makes (parts of) your code understandable to interested but non-technical parties. It offers other advantages as well: by embedding the language and rules of the domain within your programming language, you can more clearly express behaviors within the domain, which will make coding easier and less buggy. With static typing, your compiler will also be able to keep you from making mistakes because it 'knows' something about the domain.

Consider this extremely simple example. We are working in a bank and as a bank we like counting money. So our bosses want us to write a function for counting money. Beforehand, a very wise bank teller, our domain expert, warns us to be careful: the bank receives two kinds of currency, euros and dollars. And obviously you cannot just combine different valutas. Instead, you have to return an overview of the amount of money in dollars and euros, separated.

So we start writing and we add some types to help us keep track of the different kinds of currencies.

```
type Dollar = {
    _type: 'dollar'
}
type Euro = {
    _type: 'euro'
}

type Currency = Dollar | Euro;

type Money = {
    amount: number,
    currency: Currency,
}
```

You are already familiar with all of this. `Currency` is a discriminated union and marks what kind of money we are dealing with. To actually create `Money`, we have two **smart constructors**. Smart constructors offer a safe and convenient way of creating values. They make sure any objects we create are actually valid, while hiding any complexity that outsiders do not need to know about. In our case, our smart constructors make sure we never try to create `Money` with an invalid `Currency`.

```
export const dollars = (amount: number): Money => ({
    amount,
    currency: {
        _type: 'dollar',
    }
});

export const euros = (amount: number): Money => ({
    amount,
    currency: {
        _type: 'euro',
    }
});
```

These are quite basic, since they do not have any checks on the amount that is passed in. If we, for example, only wanted to accept non-negative amounts, we could throw an error (which is often the way OOP prevents illegal creation of objects) or return an `Option<Money>` which would be empty if the amount was invalid. By forcing other parts of the code to use this constructor, every other method has to 'acknowledge' that we might not have a valid amount of money to use. That way, the signature of our functions has once again become more honest.

Note that we did not name our functions `createMoneyAsDollars` or something similar. We are exposing both these functions and want them to have outsider-friendly names.

Next, our money counting function:

```
export function count(...ms: Money[]) {
    return ms.reduce((acc, curr) => {
        if (curr.currency._type === 'dollar') {
            acc.dollars += curr.amount;
        } else {
            acc.euros += curr.amount;
        }
        return acc;
    }, {dollars: 0, euros: 0});
}
```

There are numerous possible improvements to this, but nevermind the implementation, because this does what we want it to do: it takes money and returns a total of dollars and euros. Now, as an example of how you would actually use this DSL:

```
import {count, dollars, euros} from "./bankdsl"

count(
    dollars(2),
    euros(5),
    dollars(1),
)
```

Which is pretty readable, right? I hope it will not surprise you when I say this returns ‘’ as a result. And because we added types to our functions, our compiler can help us a bit. We cannot create a dollar with a string parameter. Nor can we add Date as a fourth element. A subtle detail is that I did not use semicolons to mark the end of my import or call to count. To non-programmers, that’s just additional noise.

A few remarks. Javascript/Typescript is not ideal as a language for a ‘fluent’ DSL, partly because it lacks infix operators. In languages like Haskell, Purescript or Kotlin, we can do something like this (after creating infix functions `count`, `dollars`, `euros` and `and`):

```
count 2 dollars and 3 euros
```

But Typescript does not support infix because Javascript does not. So our DSLs will, unfortunately, be a bit less readable.

Also note that the difference between the above and what we were doing in previous chapters is subtle. You might say the devil is in the details. We are thinking even more about naming. What will make our code easy to understand by outsiders and non-technical people? We are thinking about ease of use as well: how can we make creating `Money` as simple as possible (using, among other things, our smart constructors)? Do we have to force the users of our language to pass an array to `count`? No, use the spread operator and they can pass in anything they like - as long as it’s money. Somewhat less important in our example is what information we expose. We export those functions that our users will be needing, like methods for creating amounts in dollars or euros. We do not see a reason for them to create a currency directly, so we hide that.

So a large application written in FP, using DSLs, would consist of code organised in modules with clear responsibilities/a single concern (like handle authentication), each of those modules offering a DSL which simplifies usage, at the same time hiding implementation details. A developer who has a need for something like authentication does not have to dive into the code - he turns to the DSL. Which is good for keeping the code manageable, organised, understandable. Btw, in FP, the mini-language we create is often called an **Algebra**. In our above example, `count` is part of the Algebra of our application.

You can write your DSL with the techniques we saw in previous chapters. Or you can turn to some more advanced techniques discussed below. And, like anything advanced that we’ve seen so far, those techniques revolve around the use of - surprise, surprise - monads.

Tagless final

I think you could summarize tagless final by saying it "raises the level of abstraction" of your program, specifically in regards to your use of monads. When you develop your DSL, your domain-specific mini-language, you will often base it on monads. Like when you have to validate or do any kind of IO. In previous chapters, we used the existing fp-ts monads like Either or Task directly as return types. In tagless final you would instead declare a new abstract return type - by convention called F. This F cannot be just about anything. It should definitely be a monad and you might limit its type even further than that. On the other hand, it is a lot less specific than using one particular monad.

The advantages you gain from this abstraction are twofold. First, by using monads as a basic element in your applications you can easily chain operations. In many of the other languages where tagless final is popular (like Scala, or Haskell) this is done using do notation. Second, your code does not depend on one specific monad, it can be run with whatever you want - as long as it fits the requirements you specified. In fact you need a so-called 'interpreter' for F to actually run the program you 'defined'. An obvious advantage of this is that you can pull in a fake interpreter when you want to test your code - replacing IO with mocks etc.

Do notation is also available within fp-ts[31], but it is less fluent and you might still prefer piping.

A relatively simple example of tagless final in fp-ts can be found in a blog post[32] by Giulio Canti (i.e. the author of fp-ts). In it he creates a new Monad, MonadIO, which has an additional capability (the `fromIO` function). This Monad is then used throughout his program.

Interesting to note, one of the people who made tagless final popular in the Scala world now seems to be advising against its use[33]. He has come to believe that it is overkill for most projects and "*premature indirection*", a level of abstraction that might sound good but will not serve any practical use.

Free monads

Free Monads are in some ways very similar to tagless final. In essence, it allows you to turn any functor (which is the 'part' of a monad that has `map`) into a monad, thereby gaining an `of` and `chain` function without any additional work.

This is in itself already quite nifty, but Free Monads can also be used to build your application as an **Abstract Syntax Tree** (**AST**). First, create your functor, which consists of the 'commands' that

[31] https://rlee.dev/writing/practical-guide-to-fp-ts-part-6
[32] https://dev.to/gcanti/functional-design-tagless-final-332k
[33] https://degoes.net/articles/tagless-horror

are important in your application (thinks of this as a DSL). Next, 'lift' it so it becomes a Free Monad. Now you can combine the commands you just created into a program. This will, yet again, look nicer in a language like Purescript or Haskell (look up some examples if you don't believe me). But it will work just the same in Typescript and it might make your application code more readable and usable.

We're not finished though: your code will not actually do anything yet! For that you need to again write an interpreter that will give meaning to each of your commands. Which again gives you an advantage when it comes to mocking. There are downsides as well, besides the added complexity. For one, the recursive nature of this solution means you might trigger a stack overflow. And even if you avoid this, using trampolining, your solution will definitely not win any performances prizes. So do not use this unless you think your domain/application (and its complexity) warrants it.

For some examples see here[34] and here[35]. These are based on a previous version of fp-ts, but most of the changes required to make these examples work are simple enough. For example, instead of calling chain on your monad, you should now use pipe to combine chains (as we've seen countless times in the previous pages) and Free is now located in the fp-ts-contrib library.

Executable and declarative encoding, algebraic design

As an aside, a related concept to all this is that of what is sometimes called executable versus declarative encoding[36]. The former is most similar to what we have done in previous chapters, while the latter is similar to the two techniques we just discussed. I.e. our DSL is just data that needs interpretation, and it can have multiple interpreters and we have the option to define more of them whenever we desire. While the former is simpler, the latter has more power and flexibility.

Another term worth mentioning is algebraic design. An algebra (mentioned earlier in this chapter) is a group of functions that work with certain types of data, with an accompanying set of laws. So we design our application starting with this algebra. Only later do we decide on a 'representation' for it. We create types, which serve as our interface, and only later write an implementation for it all. Once again, the idea is to first clearly model our domain. The actual implementation is of no concern yet - in fact, working on it might distract us. This is similar to TDD, where we focus on working on *what* we want to accomplish through writing tests, before we turn to the *how*.

So as this blog post[37] puts it: "*Most functional code doesn't solve problems directly. Instead, it creates models of solutions to problems, which are later executed (or interpreted).*"

Summary

In this brief chapter, we discussed how DSLs can improve your code, making it easier to use and maintain. And for smaller applications, creating a DSL using functions is often good enough. But

[34] https://gist.github.com/gcanti/087a190fe0090210b82939f4bf3a1531
[35] https://dev.to/ybogomolov/monadicmonday-compilation-may-5d3c
[36] https://degoes.net/articles/functional-design
[37] https://degoes.net/articles/functional-design

there are several more advanced FP techniques out there, including two we discussed here, Tagless Final and Free Monads, which might be useful in larger applications, though one should take into account downsides like added complexity and possible lesser performance.

Conclusion

What we learned

Functional Programming (or FP) is a programming paradigm which focuses on the use of functions as the primary building blocks for creating applications, as opposed to Object Oriented Programming (OOP), with its focus on classes that encapsulate both state and behavior. FP has a strong mathematical background, is popular in academic circles and has been growing in popularity among professional programmers.

But as a paradigm, Object Oriented Programming still reigns supreme, with many widely used languages geared towards OOP, for example C++, C# and Java. Hybrid languages like Kotlin and Swift make Functional Programming an option, but they still only represent a minority of use cases in the industry. Meanwhile Javascript, despite shortcomings, treats functions as first-class citizens, which enables us to write code with a very functional flavour. And with Typescript we get a useful and powerful compiler with no runtime overhead.

Still, the basic language constructs of Javascript are insufficient to enable 'real' Functional Programming by themselves. So we have to turn to a library. The one we focussed on in this book is fp-ts. It is powerful and feature-rich, well-maintained and accompanied by multiple 'helper' libraries. Documentation is more than adequate, with blog posts going into more depth for some topics. One downside is that it is not exactly beginner-friendly, a shame because it is an excellent framework for writing production-worthy functional code. Which is why this book set out to help beginners understand the basic concepts of FP and use fp-ts for writing projects.

We started with a discussion of the basics of Functional Programming: the preference for pure functions, immutability and recursiveness. The way we can use composition to create functionality, making our code cleaner, simpler, more reusable. Once that was done, we turned our attention to monads and the important role they play in building applications. In the subsequent chapters, we used our functional techniques to build example projects, starting from simple to a bit more complex. Along the way, bit by bit, we expanded our knowledge of FP and the monads from fp-ts. We also showed how to rewrite an existing application in a more functional style and briefly discussed some more advanced material like DSLs, tagless final and free monads. Hopefully, the material contained in the previous chapters will suffice in enabling you to use fp-ts in your spare-time and professional projects.

Where to go from here

I guess this is another 'exercises for the reader' section right at the end of the book. Some suggestions. If you have been skimming through this book and it piqued your interest, you might consider going

through it again in more detail. See if you understand everything and if you can write the code yourself. Note that fp-ts is still actively being developed, which might necessitate minor changes (for example, while writing this book I noticed StateReaderTaskEither's `evalState` method had become deprecated).

If you have already done this, the question becomes more open-ended. For learning more about the fundamentals of Functional Programming, you should probably look into some 'real' FP languages like Haskell or Clojure. There are numerous videos on YouTube with discussions on category theory if you want to learn more about the mathematical foundations of FP. If you want to get into the action with Typescript, try some of the things you learned here in a pet project. Maybe thrown in some AWS services, now that you know how to create them out of a simple YAML file. If everything goes well, you could use FP next time a small, greenfield project at work comes along.

Finally, if any of the other generic subjects that were briefly mentioned in this book (like DDD or TDD) were not familiar to you, go check them out! The great curse and blessing of IT is that there is always a lot more to discover about any topic. So, paraphrasing Werner Vogels, now go explore!

Appendix A: Additional Concepts

This appendix contains a few concepts that were not discussed in the main text of this book, but could still prove useful.

Writer

Writer is another classic monad. Whereas Reader was used to retrieve values at the right time, Writer is used to store them away. The most common example use case for Writer is gathering log messages.

These

These has some similarities with the Either Monad in that it has a left and a right value. However, whereas Either stops its left value from being handled, These processes both its left, its right or both at the same time.

```
const result = pipe(
    T.both('left', 0),
    T.bimap(v => v.toUpperCase(), v => v + 1),
    T.map(v => v * 2),
    T.mapLeft(v => v + ' world'),
    T.swap
);

console.log(T.toTuple(0, 'default')(result)); (1)
```

(1) This will print `[2, 'LEFT world']`. Note the `bimap` which is used to transform both the left and right value, the `mapLeft` which only changes the left and the `swap`, which swaps the two values, hence the reversed order of the resulting tuple. The fp-ts library also has a **TaskThese**, combining the powers of Task and These.

Swap is also available for monads like TaskEither. In something like monitoring, an error, normally a left, might be a good thing that you want to process. In which case you can put it in a right.