ESSENTIAL FUNCTIONAL-FIRST F#

SUCCINCT | ROBUST | PERFORMANT

IAN RUSSELL

Essential Functional-First F#

Ian Russell

This book is for sale at http://leanpub.com/essential-fsharp

This version was published on 2022-05-22

Leanpub

Contents

Preface

This is Preview 1 of the revised edition of this book. Chapters 1-4 are ready for review.

This book is written in English. As a result, there may appear to be spelling mistakes for those who have their dictionaries set to American.

A reviewers job is to advise on where the book can be improved as well as pointing out errors. The easiest way to do this is to open the .pdf file in Adobe Acrobat Reader, use the Comments system, and email the modified .pdf back to me.

This revised edition of the book will be released in the second week of July 2022. It will be available as a free download from LeanPub.

Who should read this book

This book is targeted at folks wanting to learn F# and assumes that the reader has no real knowledge of F# or functional programming. Some programming experience, particularly of C# or VB.NET, may be useful but not absolutely necessary.

Thanks

Thanks to the reviewers of the original release of the book (Yeray Cabello, Emiliano Conti, Martin Fuß, Erle Granger II, Viacheslav Koryagin, and Cédric Rup) without whom it would have been a longer and more tortuous delivery.

Thanks to Dustin Moris Gorski for Giraffe[1], Krzysztof Cieślak and Chet Husk for their work on ionide-fsharp[2], the various people who have worked on the language and compilers over the years, and finally, last but not least, Don Syme[3] for creating the wonder that is the F# language. All Hail Don Syme!

About the author

Ian Russell has over 25 years experience as a software developer in the UK. He has held many technical roles over the years but made the decision many years ago that he could do most good by remaining 'just a software developer'. Ian works remotely from the UK for Trustbit, a software

[1]https://github.com/giraffe-fsharp/Giraffe
[2]https://ionide.io/
[3]https://twitter.com/dsymetweets

solutions provider based in Vienna, on a cloud-based GPS aggregator for a logistics company written mostly in F#. Ian's .NET journey started with C# 1.1 in 2003 and he started playing in his own time with F# in 2010. He has been a regular speaker at UK user groups and conferences for over 11 years.

Introduction

This isn't a book primarily about Functional Programming. You are not going to learn about Category Theory, Lambda Calculus or even Monads. You will learn things about functional programming but only as a by-product of how we solve problems with F#. What you will learn is how F#'s functional-first approach to programming empowers everyone to write succinct, robust and performant code[4]. You will learn a lot of new terminology as you read this book but only enough to understand the features of F# that allow us to solve real-world problems with succinct and concise code. What you will take away most from this book is an understanding of how F# supports us by making it easier for us to write code the F# way than it is to be a functional programming purist.

Why Choose F#?

For many historical reasons, functional programming has been viewed as difficult to learn and not very practical for most .NET developers. This book is designed to help dispel those impressions. A question that I get asked repeatedly is:

> What is F# good for?

Whilst it's great for many things like web programming, cloud programming, Machine Learning, AI, and Data Science, the most accurate answer comes from a long-standing F# Community member Dave Thomas[5]:

> "F# is good for programming"

F# is a general-purpose language for the .NET platform along with C# and VB.NET. F# has been bundled with Visual Studio, and now the .NET SDK, since 2010. The language has been quite stable since then. Code written in 2010 is not only still valid today but is also likely to be stylistically similar to current standards.

Why would you choose F# over another language? I noticed a question from cancelerx[6] on Twitter:

> "I would like to do a lightning talk on F#. My team is primarily PHP and Python devs. What should I showcase? I want to light a spark."

My response was:

[4] https://fsharp.org/
[5] https://twitter.com/7sharp9_
[6] https://twitter.com/ndy40

> The expressive type system, composition with |>, pattern matching, collections, and the REPL.

I've been thinking about my answer and whilst I still think that the list is a good one, I think that I should have said that it is how the various features work together that make F# so special, not the individual features themselves.

This is a book about functional-first programming in F#. F# isn't a pure functional language which means that it supports other paradigms too such as imperative and object programming. My view of the choices F# has made in its language design can be summed up quite easily:

> I enjoy functional programming but I like programming in F# even more.

I'm not alone in feeling like this about F#. I hope that this book provides a useful step on a similar journey to the one that I've been on.

What does this book cover?

This book covers the core features and practices that a developer needs to know to work effectively on the types of F# Line of Business (LOB) applications we work on at Trustbit. It starts with the basics of type and function composition, pattern matching, and testing, and ends with a worked example of a simple website and API built using the wonderful Giraffe[7] library.

The original source for this book is two series of blog posts that I wrote on the Trustbit blog[8] in 2020/21. They have been significantly re-written to clarify and expand the explanations, to ensure that the code works on VS Code plus the ionide F# extension, and to take advantage of some new F# features introduced in versions 5 and 6.

During the course of reading this book, you are going to be introduced to a wide range of features that are the essentials of functional-first F#:

- F# Interactive (FSI)
- Algebraic Type System
  - Records
  - Discriminated Unions
  - Tuples
- Pattern Matching
  - Active Patterns
  - Guard Clauses
- Let bindings

[7]https://github.com/giraffe-fsharp/Giraffe
[8]https://trustbit.tech/blog

- Immutability
- Functions
  - Pure
  - Higher Order
  - Currying
  - Partial Application
  - Pipelining
  - Signatures
  - Composition
- Unit
- Effects
  - Option
  - Result
  - Task/Async
- Computation Expressions
- Collections
  - Lists
  - Sequences
  - Arrays
  - Sets
- Recursion
- Object Programming
  - Class types
  - Interfaces
  - Object Expressions
- Exceptions
- Unit Tests
- Namespaces and Modules

You will also start the journey towards writing APIs and websites in Giraffe.

Setting up your environment

1. Install F#. The best and simplest way to do this is to install the latest .NET SDK[9] (6.0.x at time of publishing).
2. Install Visual Studio Code (VS Code).
3. Install the ionide F# extension for VS Code. You may need to reload the environment. The installation page for the extension will tell you.
4. Create a folder for the book and then folders for each book chapter to hold the code you will write.

[9]https://dotnet.microsoft.com/download

F# Interactive (FSI)

If you are coming to F# from C#, the F# Interactive window will be something new. It appears in the Terminal window. In addition to a dotnet terminal, you will also have access to an F# Interactive window. You can type directly into FSI but we will send code from our files directly to FSI to compile and run.

> We run code in FSI by selecting the code that we want to run and pressing ALT + ENTER. The code will be copied into FSI, compiled, and run. If you don't change that code, you can reuse it later as FSI keeps your compiled code in memory.

Script Files vs Code Files

F# supports two types of file: Script (.fsx) and Code (.fs). We will use both types throughout this book. The primary differences in VS Code are that only .fs files are compiled into the output .exe or .dll and .fsx files are primarily used to try things out with F# Interactive.

F# code from any type of file can be run in F# Interactive by sending it using ALT + ENTER.

> There is a command line version of FSI that can be used to run script files but that is outside the scope of this book.

Getting to know VS Code and Ionide

Whilst the VS Code and ionide environment is not as fully featured as the full IDEs (Visual Studio and Jet Brains Rider), they are not devoid of features. You don't actually need that many features to easily work on F# codebases, even large ones.

Compositional-IT[10] has released a number of videos on YouTube that give an introduction to VS Code + F# shortcuts[11].

Ionide[12] is much more than just the F# extension for VS Code. Take some time to have a look at what they offer to help your F# experience.

F# Software Foundation

You should join the F# Software Foundation[13]; It's free. By joining, you will be able to access the dedicated Slack channel where there are excellent tracks for beginners and beyond.

[10] https://www.compositional-it.com/
[11] https://www.youtube.com/playlist?list=PLlzAi3ycg2x27lPUwp3Z-M2m44n681cED
[12] https://ionide.io/index.html
[13] https://foundation.fsharp.org/join

One Last Thing

This book contains lots of code. It will be tempting to copy and paste the code from the book but you may learn more by actually typing the code out as you go. I definitely learn more by doing it this way.

The time for talking has finished. Now it is time to dive into some F# code. In chapter 1 we will work through a common business use case using F#.

1 - A simple Domain Modelling Exercise

This book will introduce you to the world of functional-first programming in F#. Rather than start with theory or a formal definition, I thought that I'd start with a typical business problem and look at how we can use some of the functional programming features of F# to solve it.

> **F# Interactive** - We will be using F# Interactive to compile and run our code in this chapter. If you are not aware of how to use this, please read the section on Setting up your environment in the Introduction.

Using VS Code, open a folder to store the code from this chapter. Add a new file called 'part1.fsx'. The .fsx file is an F# script file. We are going to use script files and run the code using F# Interactive rather than creating and running a console application via the dotnet CLI.

The Problem

This problem comes from a post by Chris Roff[14] where he looks at using F# and Behaviour Driven Development[15] together.

```
Feature: Applying a discount
Scenario: Eligible Registered Customers get 10% discount when they spend £100 or more

Given the following Registered Customers
|Customer Id|Is Eligible|
|John       |true       |
|Mary       |true       |
|Richard    |false      |

When <Customer Id> spends <Spend>
Then their order total will be <Total>

Examples:
|Customer Id|   Spend|   Total|
|Mary       |   99.00|   99.00|
```

[14] https://medium.com/@bddkickstarter/functional-bdd-5014c880c935
[15] https://cucumber.io/docs/bdd/

```
|John       |  100.00|   90.00|
|Richard    |  100.00|  100.00|
|Sarah      |  100.00|  100.00|
```

Along with some examples showing how you can verify that your code is working correctly, there are a number of domain-specific words and concepts that we may want to represent in our code. We will start with a simple but naive solution and then we'll see how F#'s types can help us make it much more domain-centric and as an added benefit, be less susceptible to bugs.

Initial Version

Along with simple datatypes like string, decimal, and boolean, F# has a powerful Algebraic Type System (ATS). At this stage, think of these types as simple data structures that you can use to compose larger data structures. The simplest type we can use to model the customer is the Tuple:

```
type Customer = string * bool * bool

let fred = "Fred", true, true

let (id, isEligible, isRegistered) = fred // id = "Fred" isEligible = true isRegiste\
red = true
```

This is not a good choice for many reasons but not having the ability to name the parts of a tuple, as C# can, is the major one. How do you know which bool field represents which concept? Tuples have many great uses in F# but this isn't one of them.

Instead, we will create a Record type that solves the property name issue. We can define our initial customer record type like this:

```
type Customer = { Id:string; IsEligible:bool; IsRegistered:bool }
```

The record type, like the tuple, is an AND type; A Customer consists of an Id which is a string value, and two boolean values called IsEligible and IsRegistered. Record types, like most of the types we will see through this book, are immutable, that is they cannot be changed once created. This means that all of the data to create a Customer record must be supplied when an instance is created.

Instead of defining the record type on a single line, we can also put each field on a separate line.

> WARNING: Tabs are not supported by F#, so your IDE/editor needs to be able to convert tabs to spaces, which most do including VS code.

```
type Customer = {
    Id : string
    IsEligible : bool
    IsRegistered : bool
}
```

If you put the fields on separate lines, you no longer need to use the semi-colon separator. Spaces between the field name and type are allowed and they do not need to be consistent.

To create an instance of a customer we would write the following **below the type definition**:

```
let fred = { Id = "Fred"; IsEligible = true; IsRegistered = true }
```

Or we can use the following style:

```
let fred = {
    Id = "Fred"
    IsEligible = true
    IsRegistered = true
}
```

By using the let keyword, we have bound the name ‘fred’ to that instance of a Customer. The F# compiler has inferred the type of fred to be Customer. You can annotate the type on the value binding if you like:

```
let fred : Customer = { Id = "Fred"; IsEligible = true; IsRegistered = true }
```

We are not going to use the *fred* binding, so you can delete that line of code.

If you have used languages like C# or Java, you may have been surprised by the strict ordering of the code. F# is like JavaScript in that the compiler works from the top down. It might seem a little odd at first but you soon get used to it. It has some really nice advantages for how we write and verify our code as well as making the compiler’s job easier. This ordering applies at the project level too, so your .fs files need to be ordered in the same way, not alphabetically.

Below the Customer type, we need to create a function to calculate the total that takes a Customer and a Spend (decimal) as input parameters and returns the Total (decimal) as output:

```
// Customer -> decimal -> decimal
let calculateTotal (customer:Customer) (spend:decimal) : decimal =
    let discount =
        if customer.IsEligible && spend >= 100.0M
        then (spend * 0.1M) else 0.0M
    let total = spend - discount
    total
```

The M suffix at the end of a number tells the compiler that the number is a decimal.

There are a few things to note about functions:

- We have used 'let' again to define the function and inside the function to define the discount and total bindings.
- There is no container, such as a class, because functions are first-class citizens.
- The return type is to the right of the input arguments.
- No return keyword is needed as the last line is returned automatically.
- Use is made of significant whitespace. Tabs are not allowed.
- We used an if expression that must return the same type for true and false routes.

> **Expressions** are used throughout F# programming since they always return an output. This makes them easy to compose with other expressions and easy to test.

The function signature is **Customer** -> **decimal** -> **decimal**. The item at the end of the signature, after the last arrow, is the return type of the function. At this stage, you should read this function signature as 'this function takes a Customer and a decimal as inputs and returns a decimal as output'. This is not strictly true as you will discover in the next chapter.

> **Function Signatures** are **very important**, as you will discover in the next chapter; Get used to looking at them.

The input parameters of the *calculateTotal* function are in curried form. We can also arrange them in tupled form:

```
// (Customer * decimal) -> decimal
let calculateTotal (customer:Customer, spend:decimal) : decimal =
    let discount =
        if customer.IsEligible && spend >= 100.0M
        then (spend * 0.1M) else 0.0M
    let total = spend - discount
    total
```

Note the change in the function signature from **Customer -> decimal -> decimal** to **(Customer * decimal) -> decimal**. The body of the function remains the same, it's just the input parameters that have changed. There are significant benefits to using the curried form for most F# functions, so that is the style that you will mostly see used in this book. We will discover some of the benefits and where the name came from in the next chapter.

The F# Compiler supports a feature called Type Inference which means that most of the time it can determine types through usage without you needing to explicitly define them. As a consequence, we can re-write the function as:

```
// Customer -> decimal -> decimal
let calculateTotal customer spend =
    let discount =
        if customer.IsEligible && spend >= 100.0M then spend * 0.1M
        else 0.0M
    spend - discount
```

The IDE should still display the function signature as **Customer -> decimal -> decimal**. I also removed the total binding as I don't think it adds anything to the readability of the function.

> Don't forget to highlight the code you've written so far and press ALT + ENTER to run it in F# Interactive (FSI).

Now create a customer from our specification below the *calculateTotal* function and run in FSI:

```
let john = { Id = "John"; IsEligible = true; IsRegistered = true }
```

Rather than write a formal test, we can use FSI to run simple verifications for us. We will look at writing proper unit tests in chapter 4. Write the following after the *john* binding:

```
let assertJohn = (calculateTotal john 100.0M = 90.0M)
```

Highlight all of the code and press ALT + ENTER to run the code in FSI. What you should see is the following:

```
val assertJohn : bool = true
```

Add in the other users and test cases from the specification in the same way:

```
let john = { Id = "John"; IsEligible = true; IsRegistered = true }
let mary = { Id = "Mary"; IsEligible = true; IsRegistered = true }
let richard = { Id = "Richard"; IsEligible = false; IsRegistered = true }
let sarah = { Id = "Sarah"; IsEligible = false; IsRegistered = false }

let assertJohn = calculateTotal john 100.0M = 90.0M
let assertMary = calculateTotal mary 99.0M = 99.0M
let assertRichard = calculateTotal richard 100.0M = 100.0M
let assertSarah = calculateTotal sarah 100.0M = 100.0M
```

Highlight the new code and press ALT + ENTER. You should see the following in FSI.

```
val assertJohn : bool = true
val assertMary : bool = true
val assertRichard : bool = true
val assertSarah : bool = true
```

Your code should now look like this:

```
type Customer = {
    Id : string
    IsEligible : bool
    IsRegistered : bool
}

let calculateTotal customer spend =
    let discount =
        if customer.IsEligible && spend >= 100.0M then spend * 0.1M
        else 0.0M
    spend - discount

let john = { Id = "John"; IsEligible = true; IsRegistered = true }
let mary = { Id = "Mary"; IsEligible = true; IsRegistered = true }
let richard = { Id = "Richard"; IsEligible = false; IsRegistered = true }
let sarah = { Id = "Sarah"; IsEligible = false; IsRegistered = false }

let assertJohn = calculateTotal john 100.0M = 90.0M
let assertMary = calculateTotal mary 99.0M = 99.0M
let assertRichard = calculateTotal richard 100.0M = 100.0M
let assertSarah = calculateTotal sarah 100.0M = 100.0M
```

Whilst this code works, I don't like boolean properties representing domain concepts. In addition, it is possible to be in an invalid state where a customer could be eligible but not registered. To prevent this, we could have added a check for customer.IsRegistered = true but it is easy to forget to do this. Instead, we will take advantage of the F# type system and make domain concepts like Registered and Unregistered explicit.

Making the Implicit Explicit

Create a new file called 'part2.fsx' and copy the final code from part1.fsx into it. We are going to modify the code in part2.fsx in this section.

Firstly, we create specific record types for Registered and Unregistered Customers.

```
type RegisteredCustomer = {
    Id : string
    IsEligible : bool
}

type UnregisteredCustomer = {
        Id : string
}
```

To represent the fact that a Customer can be either Registered or Unregistered, we will use another of the built-in types in the Abstract Type System: the Discriminated Union (DU). We define the Customer type like this:

```
type Customer =
    | Registered of RegisteredCustomer
    | Guest of UnregisteredCustomer
```

This reads as "a customer can either be a Registered customer of type RegisteredCustomer or a Guest of type UnregisteredCustomer". The items are called Union Cases and consist of a Case Identifier, in this example Registered/Guest, and some optional Case Data. You can optionally attach any type or mixture of types to a union case.

Discriminated unions are closed sets, so only the cases described in the type definition are available. The only place that cases can be defined is in the type definition.

The easiest way to understand a discriminated union is to use them! We have to make changes to the users that we have defined. Firstly the UnregisteredCustomer:

```
// Guest of UnregisteredCustomer
let sarah = Guest { Id = "Sarah" }
```

Look at how the definition in the discriminated union compares to the case definition.

Now let's make the required changes to the RegisteredCustomer bindings:

```
// Registered of RegisteredCustomer
let john = Registered { Id = "John"; IsEligible = true }
let mary = Registered { Id = "Mary"; IsEligible = true }
let richard = Registered { Id = "Richard"; IsEligible = false }
```

Changing the Customer type to a discriminated union from a record also has an impact on the *calculateTotal* function. We will modify the discount calculation using another F# feature: Pattern Matching using a match expression:

```
let calculateTotal customer spend =
    let discount =
        match customer with
        | Registered c ->
            if c.IsEligible && spend >= 100.0M then spend * 0.1M else 0.0M
        | Guest _ -> 0.0M
    spend - discount
```

We are using a match expression to determine the case of the Customer.

> **Expressions** produce an output value. Pretty much everything in F# including values, functions, and control flows is an expression. This allows them to be easily composed together.

To understand what the pattern match is doing, compare the match 'Registered c' with how we constructed the users: Registered { Id = "John"; IsEligible = true }. In this case, 'c' is a placeholder for the Customer case data. The underscore in the Guest pattern match is a wildcard and implies that we don't need access to that case data. Pattern matching against discriminated unions is exhaustive. If you don't handle every case, you will get a warning from the compiler saying 'incomplete pattern match'.

We can simplify the logic with a guard clause but it does mean that we need to account for non-eligible Registered customers otherwise the match is incomplete:

```
let calculateTotal customer spend =
    let discount =
        match customer with
        | Registered c when c.IsEligible && spend >= 100.0M -> spend * 0.1M
        | Registered _ -> 0.0M
        | Guest _ -> 0.0M
    spend - discount
```

We can simplify the last two matches using a wildcard like this:

```
let calculateTotal customer spend =
    let discount =
        match customer with
        | Registered c when c.IsEligible && spend >= 100.0M -> spend * 0.1M
        | _ -> 0.0M
    spend - discount
```

> Be careful when using the wildcard like this because it may prevent the compiler from warning you of additions to the discriminated union and could result in unexpected outcomes.

The tests don't need to change.

This is much better than the naive version we had before. It's easier to understand the logic and you can no longer create data in an invalid state. Does it get better if we make Eligibility explicit as well? Let's see!

Going Further

Create a new file called 'part3.fsx' and copy the final code from part2 into it. We are going to modify the code in part3.fsx in this section.

Now we are going to make eligibility a real domain concept. We remove the IsEligible flag from RegisteredCustomer and add EligibleRegistered to the Customer type.

```
type RegisteredCustomer = {
    Id : string
}

type UnregisteredCustomer = {
    Id : string
}

type Customer =
    | EligibleRegistered of RegisteredCustomer
    | Registered of RegisteredCustomer
    | Guest of UnregisteredCustomer
```

We need to make a change to our function.

```
let calculateTotal customer spend =
    let discount =
        match customer with
        | EligibleRegistered _ when spend >= 100.0M -> spend * 0.1M
        | _ -> 0.0M
    spend - discount
```

We no longer need to test for IsEligible and we also no longer need access to the instance, so we can replace the ‘c’ with an underscore (wildcard).

We make some minor changes to our helpers.

```
let john = EligibleRegistered { Id = "John" }
let mary = EligibleRegistered { Id = "Mary" }
```

Run your code in FSI to check all is still OK.

The state of our code after all of our improvements is:

```
type RegisteredCustomer = {
        Id : string
}

type UnregisteredCustomer = {
    Id : string
}

type Customer =
    | EligibleRegistered of RegisteredCustomer
    | Registered of RegisteredCustomer
    | Guest of UnregisteredCustomer

let calculateTotal customer spend =
    let discount =
        match customer with
        | EligibleRegistered _ when spend >= 100.0M -> spend * 0.1M
        | _ -> 0.0M
    spend - discount

let john = EligibleRegistered { Id = "John" }
let mary = EligibleRegistered { Id = "Mary" }
let richard = Registered { Id = "Richard" }
let sarah = Guest { Id = "Sarah" }
let assertJohn = calculateTotal john 100.0M = 90.0M
let assertMary = calculateTotal mary 99.0M = 99.0M
let assertRichard = calculateTotal richard 100.0M = 100.0M
let assertSarah = calculateTotal sarah 100.0M = 100.0M
```

I think that this is a big improvement over where we started as readability has improved and we have prevented getting into invalid states. However, we can still do better by replacing primitives with domain concepts. We will revisit this in a later chapter. We will also look at unit testing with XUnit where we can make use of the helpers and assertions we've already written.

Now that the RegisteredCustomer and UnregisteredCustomer records are really simple, we could remove them and add the string Id value directly to the case values in the Customer discriminated union:

```
type Customer =
    | EligibleRegistered of Id:string
    | Registered of Id:string
    | Guest of Id:string

let calculateTotal customer spend =
    let discount =
        match customer with
        | EligibleRegistered _ when spend >= 100.0M -> spend * 0.1M
        | _ -> 0.0M
    spend - discount

let john = EligibleRegistered "John"
let mary = EligibleRegistered "Mary"
let richard = Registered "Richard"
let sarah = Guest "Sarah"

let assertJohn = calculateTotal john 100.0M = 90.0M
let assertMary = calculateTotal mary 99.0M = 99.0M
let assertRichard = calculateTotal richard 100.0M = 100.0M
let assertSarah = calculateTotal sarah 100.0M = 100.0M
```

I chose not to do this as it's likely that the RegisteredCustomer case data would contain more data than we currently see. That doesn't stop you from doing it only to the UnregisteredCustomer record type.

This was one approach to modelling this simple domain but we can do it in a few other styles as well.

Alternative Approaches (1 of 2)

Create a new file called 'part4.fsx' and copy the final code from part3.fsx into it. We are going to modify the code in part4.fsx in this section.

An alternative approach would be to use a discriminated union as a type in a property of a record type:

```
type CustomerType =
    | Registered of IsEligible:bool
    | Guest

type Customer = { Id:string; Type:CustomerType }

// Customer -> decimal -> decimal
let calculateTotal customer spend =
    let discount =
        match customer.Type with
        | Registered (IsEligible = isEligible) when isEligible && spend >= 100.0M ->\
 spend * 0.1M
        | _ -> 0.0M
    spend - discount

let john = { Id = "John"; Type = Registered (IsEligible = true) }
let mary = { Id = "Mary"; Type = Registered (IsEligible = true) }
let richard = { Id = "Richard"; Type = Registered (IsEligible = false) }
let sarah = { Id = "Sarah"; Type = Guest }

let assertJohn = calculateTotal john 100.0M = 90.0M
let assertMary = calculateTotal mary 99.0M = 99.0M
let assertRichard = calculateTotal richard 100.0M = 100.0M
let assertSarah = calculateTotal sarah 100.0M = 100.0M
```

This is perfectly valid but it means introducing language that isn't in our current understanding of the domain.

We can simplify the filter like this:

```
let calculateTotal customer spend =
    let discount =
        match customer.Type with
        | Registered (IsEligible = true) when spend >= 100.0M -> spend * 0.1M
        | _ -> 0.0M
    spend - discount
```

It solves the illegal state issue nicely but I don't like having to invent names. I much prefer to make my code as domain-centric as possible.

Alternative Approaches (2 of 2)

Create a new file called 'part5.fsx' and copy the final code from 'part2.fsx' into it. We are going to modify the code in 'part5.fsx' in this section.

Remember when we discounted tuples earlier as they don't allow the individual parts to be named? Discriminated unions have their own data structures that solve that.

> Although they look like tuples, the data structures of discriminated unions are not tuples.

Let's start by modifying the Customer type and the *calculateTotal* function:

```
type Customer =
    | Registered of Id:string * IsEligible:bool
    | Guest of Id:string

// Customer -> decimal -> decimal
let calculateTotal customer spend =
    let discount =
        match customer with
        | Registered (id, isEligible) when isEligible && spend >= 100.0M -> spend * \
0.1M
        | _ -> 0.0M
    spend - discount
```

You don't need to touch the asserts but we do need to change the let bindings to support the new Customer data structure:

```
let john = Registered (Id = "John", IsEligible = true)
let mary = Registered (Id = "Mary", IsEligible = true)
let richard = Registered (Id = "Richard", IsEligible = false)
let sarah = Guest (Id = "Sarah")
```

Highlight all of the code in this script file and run it in FSI to verify that the asserts all return true.

If we don't need the value of the Id property, we can use a wildcard to ignore it:

```
// Customer -> decimal -> decimal
let calculateTotal customer spend =
    let discount =
        match customer with
        | Registered (_, isEligible) when isEligible && spend >= 100.0M -> spend * 0\
.1M
        | _ -> 0.0M
    spend - discount
```

It is also possible to simplify the filter as we have done in previous versions:

```
// Customer -> decimal -> decimal
let calculateTotal customer spend =
    let discount =
        match customer with
        | Registered (IsEligible = true) when spend >= 100.0M -> spend * 0.1M
        | _ -> 0.0M
    spend - discount
```

If you need the value of the Id property, we can ask for it at the same time as applying the IsEligible = true filter:

```
// Customer -> decimal -> decimal
let calculateTotal customer spend =
    let discount =
        match customer with
        | Registered (Id = id; IsEligible = true) when spend >= 100.0M -> spend * 0.\
1M
        | _ -> 0.0M
    spend - discount
```

If you are looking at some of the filters we've written and are wondering if they could be extracted out to some kind of re-usable code, then you are in luck! The feature you are looking for is called Active Patterns. We will spend a whole chapter looking at them later in the book but this is an example of one being used in this version our code:

```
// Customer -> unit option
let (|IsEligible|_|) customer =
    match customer with
    | Registered (IsEligible = true) -> Some ()
    | _ -> None

// Customer -> decimal -> decimal
let calculateTotal customer spend =
    let discount =
        match customer with
        | IsEligible when spend >= 100.0M -> spend * 0.1M
        | _ -> 0.0M
    spend - discount
```

You will meet Option, Some, None, and Unit in chapter 3 and Active Patterns in chapter 7. I just wanted to give you a sneak preview of some of the goodies you are going to encounter later in the book.

My guiding principle for domain modelling is to be as domain-centric as possible, whilst preventing as many illegal states as I can. Modelling is an inexact task at best. Try out a number of versions to see if there is a better way than you first thought of.

Summary

In this chapter, we have combined types in many possible ways to solve a simple business problem. The types available in the F# Algebraic Type System, despite being simple AND (record and tuple) and OR (discriminated union) structures, can be combined together to build complicated data structures. Even more importantly, it can build incredibly expressive structures that model almost any kind of domain.

Despite the relative simplicity of the code we have created, we have covered quite a lot in this chapter:

- F# Interactive (FSI)
- Algebraic Type System
  - Tuples
  - Record Types
  - Discriminated Union
  - Type Composition
- Pattern Matching
  - Match Expression
  - Guard clause

- Let bindings
- Functions
- Function Signatures

In the next chapter, we will start to look at function composition: Composing bigger functions out of smaller ones.

Postscript

To illustrate the portability of the functional programming concepts we have covered in this post, one of my colleagues, Daniel Weller, wrote a Scala version of one of the solutions:

```
sealed trait Customer

case class Registered(id : String) extends Customer
case class EligibleRegistered(id : String) extends Customer
case class Guest(id: String) extends Customer

def calculateTotal(customer: Customer)(spend: Double) = {
    val discount = customer match {
        case EligibleRegistered(_) if spend >= 100.0 => spend * 0.1
        case _ => 0.0
    }
    spend - discount
}

val john = EligibleRegistered("John")
val assertJohn = (calculateTotal (john) (100.0)) == 90.0
```

As you can see, it is fairly easy to see our solution in this unfamiliar Scala code.

You can find his code here -> https://gist.github.com/frehn

2 - Function Composition

In the last chapter, we learned about some of the core features of F#. In this chapter, we are going to concentrate on another, the one that provides part of the name of the programming paradigm: Functions.

Functions in F# are fundamentally very simple:

> Functions have one simple rule: They take one input and return one output.

In this chapter, we are going to concentrate on a special subset of functions: Pure functions.

> A **Pure Function** has the following rules:
>
> - Given the same input, it will always return the same output.
> - Produces no side effects.
>
> **Side Effects** are activities such as talking to a database, sending email, handling user input, and random number generation or using the current date and time. It is highly unlikely that you will ever write a program that is free of side effects.

Functions that satisfy these rules have many benefits: They are easy to test, are cacheable, and are parallelizable.

In the previous chapter, we wrote a function that had two input parameters. Later in this chapter, I will show you why that fact and the one input parameter rule for functions are not actually conflicting. You will also see why understanding function signatures is very important.

You can do a lot in a single function but you can have better code reusability by combining smaller functions together: We call this Function Composition.

Theory

The composition of two functions relies on the output of the first one matching the input of the next.

If we have two functions (f1 and f2) that look like this pseudocode:

```
// The ' implies a generic (any) type
f1 : 'a -> 'b
f2 : 'b -> 'c
```

As the output of f1 matches the input of f2, we can combine them together to create a new function (f3):

```
f3 : f1 >> f2 // 'a -> 'c
```

Treat the >> operator as a general-purpose composition operator for any two functions.

What happens if the output of f1 does not match the input of f2?:

```
f1 : 'a -> 'b
f2 : 'c -> 'd
where 'b <> 'c
```

To resolve this, we would create an adaptor function, or use an existing one, that we can plug in between f1 and f2:

```
f3 : 'b -> 'c
```

After plugging f3 in, we get a new function f4:

```
f4 : f1 >> f3 >> f2 // 'a -> 'd
```

Any number of functions can be composed together in this way.

Let's look at a concrete example.

In Practice

I've taken and slightly simplified some of the code from Jorge Fioranelli's excellent F# Workshop[16]. Once you've finished this chapter, I suggest that you download the workshop (it's free!) and complete it.

This example has a simple record type and three functions that we can compose together because the function signatures match up.

[16]http://www.fsharpworkshop.com/

```
type Customer = {
    Id : int
    IsVip : bool
    Credit : decimal
}

// Customer -> (Customer * decimal)
let getPurchases customer =
    let purchases = if customer.Id % 2 = 0 then 120M else 80M
    (customer, purchases)

// (Customer * decimal) -> Customer
let tryPromoteToVip purchases =
    let (customer, amount) = purchases
    if amount > 100M then { customer with IsVip = true }
    else customer

// Customer -> Customer
let increaseCreditIfVip customer =
    let increase = if customer.IsVip then 100M else 50M
    { customer with Credit = customer.Credit + increase }
```

There are a few things in this sample code that we haven't seen before.

The *getPurchases* function returns a tuple. Tuples can be used for transferring small chunks of data around, generally as the output from a function. Notice the difference between the definition of the tuple (Customer * decimal) and the usage (customer, amount) when we decompose it into its constituent parts.

The other new feature is the copy-and-update record expression. This allows you to create a new record instance based on an existing record, usually with some modified data. Records and their properties are immutable by default.

These are some of the ways that F# supports to compose functions together:

```
// Composition operator
let upgradeCustomerComposed = // Customer -> Customer
    getPurchases >> tryPromoteToVip >> increaseCreditIfVip

// C# style
let upgradeCustomerNested customer = // Customer -> Customer
    increaseCreditIfVip(tryPromoteToVip(getPurchases customer))

// Procedural
```

```
let upgradeCustomerProcedural customer = // Customer -> Customer
    let customerWithPurchases = getPurchases customer
    let promotedCustomer = tryPromoteToVip customerWithPurchases
    let increasedCreditCustomer = increaseCreditIfVip promotedCustomer
    increasedCreditCustomer

// Forward pipe operator
let upgradeCustomerPiped customer = // Customer -> Customer
    customer
    |> getPurchases
    |> tryPromoteToVip
    |> increaseCreditIfVip
```

They all have the same function signature and will produce the same result with the same input.

The *upgradeCustomerPiped* function uses the forward pipe operator (|>). It is equivalent to the *upgradeCustomerProcedural* function but without having to specify the intermediate values. The value from the line above gets passed down as the last input argument of the next function. The difference between the >> and |> is that the function composition operator sits between two functions whereas the forward pipe sits between a value and a function.

> Use the forward pipe operator (|>) as your default style.

It is quite easy to verify the output of the upgrade functions using FSI.

```
let customerVIP = { Id = 1; IsVip = true; Credit = 0.0M }
let customerSTD = { Id = 2; IsVip = false; Credit = 100.0M }

let assertVIP =
    upgradeCustomerComposed customerVIP = {Id = 1; IsVip = true; Credit = 100.0M }
let assertSTDtoVIP =
    upgradeCustomerComposed customerSTD = {Id = 2; IsVip = true; Credit = 200.0M }
let assertSTD =
    upgradeCustomerComposed { customerSTD with Id = 3; Credit = 50.0M } = {Id = 3; I\
sVip = false; Credit = 100.0M }
```

Record types use **structural equality** which means that if the records look the same, they are equal.

> If two records contain the same data, they have structural equality through the equality operator (=).

Try replacing the *upgradeCustomerComposed* function with any of the other three functions in the asserts to confirm that they produce the same results in FSI.

Unit

All functions must have one input and one output. To solve the problem of a function that doesn't require an input value or produce any output, F# has a special type called unit.

> Any function taking unit as the only input or returning it as the output is causing side effects

```
open System

// unit -> System.DateTime
let now () = DateTime.UtcNow

// 'a -> unit
let log msg =
    // Log message or similar task that doesn't return a value
    ()
```

Unit appears in the function signature as unit but in code, you will use ().

If you call the *now* function without the unit () parameter you will get a result similar to this:

```
val it : (unit -> DateTime) = <fun:it@30-4>
```

This looks really strange but what you actually get is a function as the output that takes unit and returns a DateTime value as shown by the signature. To execute the function, you need to supply the input parameter. This is an important F# feature called Partial Application, which we will explain later in this chapter.

If you forget to add the unit parameter '()' when you define the binding, you get a fixed value from the first time the line is executed:

```
let fixedNow = DateTime.UtcNow

let theTimeIs = fixedNow
```

Wait a few seconds and run the *theTimeIs* binding again and you'll notice that the date and time haven't changed from when the binding was first used.

> A function binding always has at least one parameter, even if it is just unit, otherwise, it is a value binding. You can tell if it's a function by whether the signature has an arrow (->) or not.

Anonymous Functions

So far, we have only dealt with named functions but we can also create functions without names, commonly called anonymous functions.

```
// int -> int -> int
let add x y = x + y

// int -> int -> int
let add x = fun y -> x + y
```

Anytime you see the fun keyword, you are looking at an anonymous function.

They are useful in many ways. This example uses scoping rules to re-use the same instance of the Random class to produce a list of 100 items between 0 and 99:

```
open System

// unit -> int
let rnd =
    let rand = Random()
    fun () -> rand.Next(100)

List.init 100 (fun _ -> rnd())
```

Multiple Parameters

At the start of the chapter, I stated that it is a rule that all functions must have one input and one output but in the last chapter, we created a function with multiple input parameters:

```
// Customer -> decimal -> decimal
let calculateTotal customer spend = ...
```

Let's write the function signature using an anonymous function:

```
Customer -> decimal -> decimal
let calculateTotal customer =
        fun spend -> (...)
```

The *calculateTotal* function takes a Customer as input and returns a function with the signature (decimal -> decimal) as output; That function takes a decimal as input and returns a decimal as output. The ability to automatically chain functions together like this is called **Currying** after Haskell Curry, a US Mathematician. It allows you to write functions that appear to have multiple input arguments but also opens the way to a very powerful functional concept called Partial Application.

Partial Application

We will use the *calculateTotal* function to illustrate how partial application works. We can ignore the actual implementation of the function as we only care about the input parameters:

```
// Customer -> decimal -> decimal
let calculateTotal customer spend = ...
```

If I only provide the first parameter, I get a function as output with signature (Decimal -> Decimal):

```
// Decimal -> Decimal
let partial = calculateTotal john
```

If we then supply the final parameter, the original function completes and returns the expected output:

```
// Decimal
let complete = partial 100.0M
```

Run the code in FSI and look at the outputs from both lines of code.

You must add the input parameters in order, in ones or multiples. Trying to add a parameter out of order will not work:

```
let doesNotWork = calculateTotal 100M // Does not compile
```

You may wonder why you would want to do this but partial application has some very important uses such as allowing the forward pipe operator (|>) we used earlier in this chapter.

Let's have a look at a simplae practical example. Create a simple function that takes the log level as a discriminated union and a string for the message:

```
type LogLevel =
    | Error
    | Warning
    | Info

// LogLevel -> string -> unit
let log (level:LogLevel) message =
    printfn "[%A]: %s" level message
    ()
```

To partially apply this function, I'm going to define a new function that only takes the log function and its level argument but not the message:

```
let logError = log Error // string -> unit
```

The name logError is bound to a function that takes a string and returns unit. So now, we can use the logError function instead:

```
let m1 = log Error "Curried function"

let m2 = logError "Partially Applied function"
```

As the return type is unit, you don't have to let bind the function to a value:

```
log Error "Curried function"

logError "Partially Applied function"
```

When you use functions that return unit in real applications, you will get warned to ignore the output. You do that like this:

```
logError "Error message" |> ignore
```

Partial application is a very powerful concept that is only made possible because of curried input parameters. It is not possible with tupled parameters as you need to supply all parameters at once:

```
type LogLevel =
    | Error
    | Warning
    | Info

// (LogLevel * string) -> unit
let log (level:LogLevel, message:string) =
    printfn "[%A]: %s" level message
    ()
```

There's so much more that we could cover here but to do that we need to cover **Higher Order functions** first. We will meet them in the next chapter.

Summary

In this chapter, we have covered:

- Pure Functions

- Anonymous Functions
- Function Composition
- Tuples
- Copy-and-update record expression
- Curried and Tupled parameters
- Currying and Partial Application

We have now covered the fundamental building blocks of programming in F#: Composition of types and functions, expressions, and immutability.

In the next chapter, we will investigate the handling of null and exceptions in F#.

3 - Null and Exception Handling

In this chapter, we will investigate how we handle nulls and exceptions in F#. In addition, we'll be extending our understanding of function composition that we looked at in the previous chapter and looking at higher order functions.

Null Handling

Most of the time, you will not have to deal with null in your F# code as it has a built-in type called Option that you will use instead. It looks very similar to this:

```
type Option<'T> =
    | Some of 'T
    | None
```

It is a discriminated union with two cases. The single tick (') before the T is the F# way of showing that T is a Generic type. As a consequence, any type can be made optional.

Create a new file in your folder called option.fsx.

Don't forget to highlight and run the code examples in this chapter in F# Interactive (FSI).

We'll start by creating a function to try to parse a string as a DateTime:

```
open System

// string -> DateTime option
let tryParseDateTime (input:string) =
    let (success, value) = DateTime.TryParse input
    if success then Some value else None
```

You can also pattern match the result of the function with a match expression instead of using an if expression:

```
// string -> DateTime option
let tryParseDateTime (input:string) =
    match DateTime.TryParse input with
    | true, result -> Some result
    | _ -> None
```

Either of these code styles is acceptable.

Run the following examples in FSI with either version of the function:

```
let isDate = tryParseDateTime "2019-08-01" // Some 01/08/2019 00:00:00

let isNotDate = tryParseDateTime "Hello" // None
```

You will see that the string that can be parsed into a valid date returns Some of the valid date and the non-date string returns None.

Another way that the Option type can be used is for optional data like a person's middle name as not everyone has one:

```
type PersonName = {
    FirstName : string
    MiddleName : string option // or Option<string>
    LastName : string
}
```

If the person doesn't have a middle name, you set it to None and if they do you set it to Some "name", as shown here:

```
let person = { FirstName = "Ian"; MiddleName = None; LastName = "Russell"}

let person2 = { person with MiddleName = Some "????" }
```

Notice that we have used the copy-and-update record expression we met in the last chapter.

Sadly, there is one area where nulls can sneak into your codebase and that is through interop with code/libraries written in other .Net languages such as C# including most of the BCL in .NET core.

Interop With .NET

If you are interacting with code written in C#, there is a chance that you will have some null issues. In addition to the Option type, F# also offers the Option module that contains some very useful helper functions to make life easier.

Let's create a null for both a Reference type and a Nullable primitive:

```
open System

// Reference type
let nullObj:string = null

// Nullable type
let nullPri = Nullable<int>()
```

Run the code in FSI to prove that they are both null.

To convert from .Net to an F# Option type, we can use the Option.ofObj and Option.ofNullable functions:

```
let fromNullObj = Option.ofObj nullObj

let fromNullPri = Option.ofNullable nullPri
```

To convert from an Option type to .Net types, we can use the Option.toObj and Option.toNullable functions.

```
let toNullObj = Option.toObj fromNullObj

let toNullPri = Option.toNullable fromNullPri
```

Run the code in FSI to show that this works correctly:

What happens if you want to convert from an Option type to something that doesn't support null but instead expects a placeholder value? You could use pattern matching as Option is a discriminated union or you can use the defaultValue function from the Option module:

```
let resultPM input =
    match input with
    | Some value -> value
    | None -> "------"

let resultDV = Option.defaultValue "------" fromNullObj
```

If the Option value is Some, then the value wrapped by Some is returned, otherwise, the default value, in these cases, "——", is returned.

You could also use the forward pipe operator (|>):

```
let resultFP = fromNullObj |> Option.defaultValue "------"

let resultFPA =
    fromNullObj
    |> Option.defaultValue "------"
```

If you use this a lot, you may find that using Partial Application might make the task more pleasurable. We create a function that takes the default but not the Option value:

```
// (string option -> string)
let setUnknownAsDefault = Option.defaultValue "????"

let result = setUnknownAsDefault fromNullObj
```

Or using the forward pipe operator:

```
let result = fromNullObj |> setUnknownAsDefault
```

As you can see, handling of null and optional values is handled very nicely in F#. If you are diligent, you should never see a NullReferenceException in a running F# application.

Handling Exceptions

Create a new file called result.fsx in your folder.

We will create a function that does simple division but returns an exception if the divisor is 0:

```
open System

// decimal -> decimal -> decimal
let tryDivide (x:decimal) (y:decimal) =
    try
        x/y
    with
    | :? DivideByZeroException as ex -> raise ex
```

Whilst this code is perfectly valid, the function signature is lying to you; It doesn't always return a decimal. The only way I would know this is by looking at the code or getting the error when the code executed. This goes against the general ethos of F# coding.

Most functional languages implement a type that offers a choice between success and failure and F# is no exception. This is an example of a potential implementation:

```
type Result<'TSuccess,'TFailure> =
    | Success of 'TSuccess
    | Failure of 'TFailure
```

Unsurprisingly, there is one built into the language (from F# 4.1) but rather than Success/Failure, it uses Ok/Error. Let's use the Result type in our tryDivide function:

```
// decimal -> decimal -> Result<decimal,exn>
let tryDivide (x:decimal) (y:decimal) =
    try
        Ok (x/y)
    with
    | :? DivideByZeroException as ex -> Error ex
```

The try with expression is used to handle exceptions. In our case, we are only expecting one specific error type: System.DivideByZeroException.

The ':?' operator is a pattern matching feature to match a specified type or subtype, in this case, if the error can be cast as System.DivideByZeroException. The 'as ex' gives us access to the actual exception instance which we then pass to the Error result. Any other exception gets passed up the call chain until something else handles it or it crashes the application, just like the rest of .NET.

Run the code in FSI to see what you get from an error and success cases:

```
let badDivide = tryDivide 1M 0M // Error "System.DivideByZeroException: Attempted to\
 divide by zero..."

let goodDivide = tryDivide 1M 1M // Some 1M
```

Next, we are going to look at how we can incorporate the Result type into the composition code we used in the last post.

Function Composition With Result

I have modified the *getPurchases* and *increaseCreditIfVip* functions to return Result types but have left the *tryPromoteToVip* function alone so that we see the impact the changes have on function composition:

```fsharp
open System

type Customer = {
    Id : int
    IsVip : bool
    Credit : decimal
}

// Customer -> Result<(Customer * decimal),exn>
let getPurchases customer =
    try
        // Imagine this function is fetching data from a Database
        let purchases =
            if customer.Id % 2 = 0 then (customer, 120M)
            else (customer, 80M)
        Ok purchases
    with
    | ex -> Error ex

// Customer * decimal -> Customer
let tryPromoteToVip purchases =
    let customer, amount = purchases
    if amount > 100M then { customer with IsVip = true }
    else customer

// Customer -> Result<Customer,exn>
let increaseCreditIfVip customer =
    try
        // Imagine this function could cause an exception
        let credit =
            if customer.IsVip then 100.0M else 50.0M
        Ok { customer with Credit = customer.Credit + credit }
    with
    | ex -> Error ex

let upgradeCustomer customer =
    customer
    |> getPurchases
    |> tryPromoteToVip // Compiler problem
    |> increaseCreditIfVip

let customerVIP = { Id = 1; IsVip = true; Credit = 0.0M }
let customerSTD = { Id = 2; IsVip = false; Credit = 100.0M }
let assertVIP =
        upgradeCustomer customerVIP = Ok {Id = 1; IsVip = true; Credit = 100.0M }
let assertSTDtoVIP =
        upgradeCustomer customerSTD = Ok {Id = 2; IsVip = true; Credit = 200.0M }
let assertSTD =
        upgradeCustomer { customerSTD with Id = 3; Credit = 50.0M } = Ok {Id = 3; IsVip = f\
alse; Credit = 100.0M }
```

There is a problem in the *upgradeCustomer* function on the call to the *tryPromoteToVip* function because the function signatures don't match up any longer. To solve this, we are going to use a functional feature called Higher Order functions.

> **Higher Order Functions** take one or more functions as arguments and/or return a function as its output.

Scott Wlaschin[17] visualises composition with the Result type as two parallel railway tracks which he calls Railway Oriented Programming (ROP), with one track for Ok and one for Error. You travel on the Ok track until you have an error and then you switch to the Error track. He defines the *tryPromoteToVip* function as a one-track function because it doesn't output a Result type and will only execute on the Ok track. This causes an issue because the output from the previous function and input to the next function doesn't match.

The first thing that we need to do is to use a match expression in an anonymous function to unwrap the tuple from the Ok part of the Result returned from the *getPurchases* function:

```
let upgradeCustomer customer =
    customer
    |> getPurchases
    |> fun result ->
        match result with
        | Ok x -> Ok (tryPromoteToVip x)
        | Error ex -> Error ex
    |> increaseCreditIfVip // Compiler problem
```

We have to return a Result from this code, so we wrap the call to *tryPromoteToVip* in an Ok.

We now do the same for the *increaseCustomerCreditIfVip* function. In this case, we don't need to add the Ok as it already returns a result:

[17]https://fsharpforfunandprofit.com/rop/

```
let upgradeCustomer customer =
    customer
    |> getPurchases
    |> fun result ->
        match result with
        | Ok x -> Ok (tryPromoteToVip x)
        | Error ex -> Error ex
    |> fun result ->
        match result with
        | Ok x -> increaseCreditIfVip x
        | Error ex -> Error ex
```

It is possible to simplify this code slightly like this:

```
let upgradeCustomer customer =
    customer
    |> getPurchases
    |> function
        | Ok x -> Ok (tryPromoteToVip x)
        | Error ex -> Error ex
    |> function
        | Ok x -> increaseCreditIfVip x
        | Error ex -> Error ex
```

Either style is fine to use.

What we need to do now is to convet our anonymous functions into named functions. We start with the tryToPromoteToVip block. Our new function has the following signature:

```
(Customer * decimal -> Customer) -> Result<Customer * decimal, exn> -> Result<Custom\
er, exn>
```

We are going to call this function *map* for reasons that will become clear later in the section:

```
// (Customer * decimal -> Customer) -> Result<Customer * decimal, exn> -> Result<Cus\
tomer, exn>
let map (tryPromoteToVip:Customer * decimal -> Customer) (result:Result<Customer * d\
ecimal, exn>) : Result<Customer, exn> =
        match result with
    | Ok x -> Ok (tryPromoteToVip x)
    | Error ex -> Error ex
```

We don't use the types specific to *tryPromoteToVip* or the result, so we can make this function take generic parameters:

```
// ('a -> 'b) -> Result<'a,'c> -> Result<'b,'c>
let map (f:'a -> 'b) (result:Result<'a, 'c>) : Result<'b, 'c> =
        match result with
    | Ok x -> Ok (f x)
    | Error ex -> Error ex
```

We can remove the types from the function and the compiler will confirm that it is generic:

```
// ('a -> 'b) -> Result<'a,'c> -> Result<'b,'c>
let map f result =
        match result with
    | Ok x -> Ok (f x)
    | Error ex -> Error ex
```

We can easily show this is true by trying the new map function with some code that matches the same pattern, such as the *tryParseDateTime* function we met earlier:

```
// ('a -> 'b) -> Result<'a,'c> -> Result<'b,'c>
let map f result =
           match result with
    | Ok x -> Ok (f x)
    | Error ex -> Error ex

// string -> DateTime option
let tryParseDateTime (input:string) =
    let success, value = DateTime.TryParse input
    if success then Some value else None

// Result<string, exn>
let getResult =
    try
        Ok "Hello"
    with
    | ex -> Error ex

// Result<DateTime option, exn>
let parsedDT = getResult |> map tryParseDateTime
```

If we use the map function in our *upgradeCustomer* function we get this:

```
let upgradeCustomer customer =
    customer
    |> getPurchases
    |> map tryPromoteToVip
    |> fun result ->
        match result with
        | Ok x -> increaseCreditIfVip x
        | Error ex -> Error ex
```

Now we do a similar thing for the *increaseCreditIfVip* function. This slightly different to the map function as we don't need to wrap the output in a result as the *increaseCreditIfVip* already does that. The function will be called *bind* for reasons we will see very soon:

```
// (Customer -> Result<Customer, exn>) -> Result<Customer, exn> -> Result<Customer, \
exn>
let bind (increaseCreditIfVip:Customer -> Result<Customer, exn>) (result:Result<Cust\
omer, exn>) : Result<Customer, exn> =
        match result with
    | Ok x -> increaseCreditIfVip x
    | Error ex -> Error ex
```

Again, this can be made generic as we don't use any of the parameters in the function:

```
// ('a -> Result<'b, 'c>) -> Result<'a, 'c> -> Result<'b, 'c>
let bind (f:'a -> Result<'b, 'c>) (result:Result<'a, 'c>) : Result<'b, 'c> =
        match result with
    | Ok x -> f x
    | Error ex -> Error ex
```

And we can remove the parameter types:

```
// ('a -> Result<'b, 'c>) -> Result<'a, 'c> -> Result<'b, 'c>
let bind f result =
        match result with
    | Ok x -> f x
    | Error ex -> Error ex
```

Let's plug the bind function into *upgradeCustomer*:

```
let upgradeCustomer customer =
    customer
    |> getPurchases
    |> map tryPromoteToVip
    |> bind increaseCreditIfVip
```

The code should now have no compiler warnings. Run the code in FSI and then the asserts to verify the code works as expected.

Written as procedural code, the *upgradeCustomer* function looks like this:

```
let upgradeCustomer customer =
    let purchasedResult = getPurchases customer
    let promotedResult = map tryPromoteToVip purchasedResult
    let increaseResult = bind increaseCreditIfVip promotedResult
    increaseResult
```

The reason for using map and bind as function names is because the Result module in F# has them built-in as it is a common requirement. Let's update the *upgradeCustomer* function to use the Result module functions rather than our own.

```
let upgradeCustomer customer =
    customer
    |> getPurchases
    |> Result.map tryPromoteToVip
    |> Result.bind increaseCreditIfVip
```

We can delete our map and bind functions as we don't need them any longer.

If you want to learn more about this style of programming, I highly recommend Scott's book Domain Modelling Made Functional[18]. Not only does it cover Railway Oriented Programming (ROP) but also lots of very useful Domain-Driven Design information. It's my favourite technical book!

Summary

We've completed another chapter and have covered a lot of additional features and concepts that build upon our existing knowledge.

- Null handling
- Option type and module
- Exception handling
- Result type and module
- Higher Order Functions

In the next chapter, we'll start looking at organising your code into projects and unit testing.

[18]https://pragprog.com/book/swdddf/domain-modeling-made-functional

4 - Organising Code and Testing

In this chapter, we will be looking at how we can use organise our code using .NET Solutions, Projects, Namespaces, and F# Modules. In addition, we'll be writing our first real unit tests in F# with XUnit.

Getting Started

Follow the instructions in the Appendix of this book to create a new .NET solution containing two projects, a console for the code and an XUnit one for the tests.

Once completed, you should see a message like the following in the tests terminal:

```
Passed!  - Failed:     0, Passed:     1, Skipped:     0, Total:     1, Duration: < 1\
 ms - MyProjectTests.dll (net5.0)
```

Open a second Terminal window and execute the following commands:

```
cd src/MyProject
```

```
dotnet run
```

You should see a message like the following in the src terminal:

```
Hello world from F#
```

Solutions and Projects

We have created a Solution to which we added two directories; src for code projects and test for test projects. The test project has to reference the code project to be able to test it.

Adding a Source File

The last thing we need to do is add a new file called *Customer.fs*. We can do this in two ways. You will find it beneficial to know them both when working on F# in VS Code. We can use either the Explorer view (top icon in the toolbar) or the F# Explorer (Look for the F# logo).

Using Explorer:

If you manually add a file, you have to manually include it to the .fsproj file as well so that F# recognises it:

1. Select the MyProject.fsproj file.
2. Click on the New File icon and name the file Customer.fs.
3. Click on the MyProject.fsproj file to edit it. Copy the '<Compile Include="Program.fs" />' entry and paste it above or below the existing entry.
4. Change the file name of the uppermost entry to Customer.fs from Program.fs.

MyProject.fsproj should now contain this:

```
<ItemGroup>
  <Compile Include="Customer.fs" />
  <Compile Include="Program.fs" />
</ItemGroup>
```

Using F# Solution Explorer:

1. Right-click on Program.fs in the MyProject folder.
2. Select the Add File Above menu item and name the file Customer.fs.

Once your new file is created, run 'dotnet build' on the project or solution from the terminal and you will get an error like this:

```
Files in libraries or multiple-file applications must begin with a namespace or modu\
le declaration, e.g. 'namespace SomeNamespace.SubNamespace' or 'module SomeNamespace\
.SomeModule'. Only the last source file of an application may omit any declaration.
```

Namespaces and Modules

Modules are how we logically organise our code in an F# project. Namespaces are used to reduce the chances of naming collisions when referencing code from other projects.

Namespaces

- Can only contain type declarations, import declarations, or modules.
- Span across multiple files.
- Module names must be unique within a namespace even across multiple files.

Modules

- Can contain anything apart from namespaces. You can even nest modules.
- You can have a top-level module instead of a namespace but it must be unique across all files in the project.

Let's look at an example of using namespaces and modules:

```
namespace MyApplication.Customer // Namespace.Namespace

open System

type Customer = {
    Name : string
}

module Domain =

    // string -> Customer
    let create (name:string) =
        { Name = name }

module Db =

    open System.IO

    // Customer -> bool
    let save (customer:Customer) =
        // Imagine this talks to a database
        ()
```

The Customer record type is used by the two modules, so it is defined in the scope of the namespace. The modules have access to the types in the namespace without having to add an import declaration.

To fix our build issue, we are going to go add a namespace to the Customer.fs file in the MyProject project. Add the namespace declaration to the top of the file:

```
namespace MyProject
```

The name can be anything you like, it doesn't have to match the project name. If you run 'dotnet build' in the terminal again, the error has been fixed.

There are other combinations of namespace and module available. Firstly, separate the namespace and module definition:

```
namespace MyApplication // Namespace

open System

module Customer = // Module

    type Customer = {
        Name : string
    }

    module Domain =

        // string -> Customer
        let create (name:string) =
            { Name = name }

    module Db =

        open System.IO

        // Customer -> bool
        let save (customer:Customer) =
            // Imagine this talks to a database
            ()
```

Notice the nesting required for the sub-modules.

We can also use the module keyword at the top level. In this case, MyApplication is a namespace and Customer, a module:

```
module MyApplication.Customer // Namespace.Module

open System

type Customer = {
    Name : string
}

module Domain =

    // string -> Customer
    let create (name:string) =
        { Name = name }
```

```
module Db =

    open System.IO

    // Customer -> bool
    let save (customer:Customer) =
        // Imagine this talks to a database
        ()
```

A good starting point is to add a namespace to the top of each file using the project name and use modules for everything else.

Do not feel pressured into creating a file per module. It's generally better to keep as much code that changes together in the same file. This means organising your code by feature/domain concept rather than by technical concept as happens in MVC for instance.

Tests

Have a look at the Tests.fs file that was created when we generated the test project:

```
module Tests

open System
open Xunit

[<Fact>]
let ``My test`` () =
    Assert.True(true)
```

One of the really nice things about F# for naming things, particularly tests, is that we can use readable sentences. When you have test errors, the name of the module and the test appear in the output.

There is generally no need for namespaces in test files as the code is not deployed.

> **WARNING**: If you use the double backtick naming style, do not use any odd chars like [,|-;.]. These used in test names causes the ionide extension to crash. [Correct: 2022-05-20]

Create a new file in the test project called CustomerTests.fs and add the following code to it:

```
namespace MyProjectTests

open System
open Xunit

module ``I can group my tests in a module and run`` =

    [<Fact>]
    let ``My first test`` () =
        Assert.True(true)

    [<Fact>]
    let ``My second test`` () =
        Assert.True(true)
```

Run the tests using the Terminal:

```
dotnet test
```

You can also use the test runner built into VS Code.

Change the second test to make it fail.

```
    [<Fact>]
    let ``My second test`` () =
        Assert.True(false)
```

Run the tests and look at the output.

Writing Real Tests

We are going to use the code from chapter 2 to write tests against.

Create a new file called Customer.fs into MyProject and write the following code into it:

```
module MyProject.Customer

type Customer = {
    Id : int
    IsVip : bool
    Credit : decimal
}

// Customer -> (Customer * decimal)
let getPurchases customer =
    let purchases = if customer.Id % 2 = 0 then 120M else 80M
    (customer, purchases)

// (Customer * decimal) -> Customer
let tryPromoteToVip purchases =
    let (customer, amount) = purchases
    if amount > 100M then { customer with IsVip = true }
    else customer

// Customer -> Customer
let increaseCreditIfVip customer =
    let increase = if customer.IsVip then 100M else 50M
    { customer with Credit = customer.Credit + increase }

// Customer -> Customer
let upgradeCustomer customer =
    customer
    |> getPurchases
    |> tryPromoteToVip
    |> increaseCreditIfVip
```

We can use the asserts from chapter 2 as the basis of our new tests:

```
let customerVIP = { Id = 1; IsVip = true; Credit = 0.0M }
let customerSTD = { Id = 2; IsVip = false; Credit = 100.0M }

let assertVIP =
        upgradeCustomer customerVIP = {Id = 1; IsVip = true; Credit = 100.0M }
let assertSTDtoVIP =
        upgradeCustomer customerSTD = {Id = 2; IsVip = true; Credit = 200.0M }
let assertSTD =
        upgradeCustomer { customerSTD with Id = 3; Credit = 50.0M } = {Id = 3; IsVip = fals\
e; Credit = 100.0M }
```

Delete the code from CustomerTests.fs and write the following code into the new file:

```
namespace MyProjectTests

open Xunit
open MyApplication.Customer

module ``When upgrading customer`` =

    let customerVIP = { Id = 1; IsVip = true; Credit = 0.0M }
    let customerSTD = { Id = 2; IsVip = false; Credit = 100.0M }

    // let assertVIP =
    //     upgradeCustomer customerVIP = {Id = 1; IsVip = true; Credit = 100.0M }
    // let assertSTDtoVIP =
    //     upgradeCustomer customerSTD = {Id = 2; IsVip = true; Credit = 200.0M }
    // let assertSTD =
    //     upgradeCustomer { customerSTD with Id = 3; Credit = 50.0M } = {Id = 3; Is\
Vip = false; Credit = 100.0M }

    [<Fact>]
    let ``should give VIP customer more credit`` () =
        let expected = { customerVIP with Credit = customerVIP.Credit + 100M }
        let upgraded = upgradeCustomer customerVIP
        Assert.Equal(expected, upgraded)

    [<Fact>]
    let ``should convert eligible STD customer to VIP`` () =
        let customer = { Id = 2; IsVip = false; Credit = 200M }
        let expected = { customer with IsVip = true; Credit = customer.Credit + 100M\
 }
        let upgraded = upgradeCustomer customer
        Assert.Equal(expected, upgraded)

    [<Fact>]
    let ``should not upgrade eligible STD customer to VIP`` () =
        let customer = { Id = 3; IsVip = false; Credit = 50M }
        let expected = { customer with Credit = customer.Credit + 50M }
        let upgraded = upgradeCustomer customer
        Assert.Equal(expected, upgraded)
```

We are using Xunit as our test framework. There are other testing and assertion libraries available that you may prefer. This is how the assertion would look if we used FsUnit:

```
// Assert.Equal(expected, upgraded)

open FsUnit

upgraded |> should equal expected
```

Add the remaining tests to replace our simple asserts:

```
    [<Fact>]
    let ``should convert eligible STD customer to VIP`` () =
        let customer = { Id = 2; IsVip = false; Credit = 200M }
        let expected = { customer with IsVip = true; Credit = customer.Credit + 100M\
 }
        let upgraded = upgradeCustomer customer
        Assert.Equal(expected, upgraded)

    [<Fact>]
    let ``should not upgrade eligible STD customer to VIP`` () =
        let customer = { Id = 3; IsVip = false; Credit = 50M }
        let expected = { customer with Credit = customer.Credit + 50M }
        let upgraded = upgradeCustomer customer
        Assert.Equal(expected, upgraded)
```

Run the tests by running ‘dotnet test’ in the tests terminal. They should pass.

Replace the asserts with the FsUnit ones and re-run the tests.

Summary

The features we used in this chapter are important if we want to make working on larger F# codebases manageable.

- Solutions, projects, namespaces, and modules
- Unit testing with Xunit
- Assertions with FsUnit

In the next chapter, we will have an initial look at collections.

5 - Introduction to Collections

In this chapter, we will investigating functional collections and their helper modules.

Before We Start

Create a new folder for this chapter.

The Basics

Add a new script file called lists.fsx for the example code we are going to going to write in this section.

F# has a hugely deserved reputation for being great for data-centric use cases like finance and data science due in a large part to the power of its support for data structures and collection handling.

There are a number of types of collections in F# that we can make use of but the three primary ones are:

- Seq - A lazily evaluated collection that is the equivalent of IEnumerable<'T>.
- Array - Great for numerics/data science. There are built-in modules for 2d, 3d and 4d arrays.
- List - Eagerly evaluated and immutable structure and data. F# specific, not same as List<'T> in .NET.

  .NET List<'T> is called ResizeArray<'T> in F#

Each of these types has a supporting module that contains a wide range of functions including the ability to convert to/from each other.

In this post we are going to concentrate on the List type and module.

Core Functionality

We will be using lists.fsx for this section. Remember to highlight the code and run it in F# Interactive (FSI) using ALT + ENTER.

Create an empty list:

```
let items = []
```

Create a list with five integers:

```
let items = [1;2;3;4;5]
```

In this case, we could also do this:

```
let items = [1..5]
```

Or we could use a List Comprehension:

```
let items = [ for x in 1..5 do x ]
```

Comprehensions are really powerful. They're also available for the other primary collection types in F#, seq and array, but we are not going to use them in this chapter.

To add an item to a list, we use the cons operator (::):

```
// head :: tail
let extendedItems = 6::items
```

The original list remains unaffected by the new item as it is immutable.

A list is made up of a head (single item) and a tail (list of items) which could be empty ([]). We can pattern match on a list to show this:

```
let readList items =
    match items with
    | [] -> "Empty list"
    | [head] -> $"Head: {head}" //Single item list
    | head::tail -> sprintf "Head: %A and Tail: %A" head tail

let emptyList = readList [] // "Empty list"
let multipleList = readList [1;2;3;4;5] // "Head: 1 and Tail: [2;3;4;5]"
let singleItemList = readList [1] // "Head: 1"
```

If we remove the pattern match for the single item list, the code still works:

```
let readList items =
    match items with
    | [] -> "Empty list"
    | head::tail -> sprintf "Head: %A and Tail: %A" head tail

let emptyList = readList [] // "Empty list"
let multipleList = readList [1;2;3;4;5] // "Head: 1 and Tail: [2;3;4;5]"
let singleItemList = readList [1] // "Head: 1 and Tail: []"
```

We can join (concatenate) two lists together:

```
let list1 = [1..5]
let list2 = [3..7]
let emptyList = []

let joined = list1 @ list2 // [1;2;3;4;5;3;4;5;6;7]
let joinedEmpty = list1 @ emptyList // [1;2;3;4;5]
let emptyJoined = emptyList @ list1 // [1;2;3;4;5]
```

As lists are immutable, we can re-use them knowing that their values/structure will never change.

We could use the List.concat function to do the same job as the @ operator:

```
let joined = List.concat [list1;list2]
```

We can filter a list using a predicate function with the signature (‘a -> bool) and the List.filter function:

```
let myList = [1..9]

let getEvens items =
    items
    |> List.filter (fun x -> x % 2 = 0)

let evens = getEvens myList // [2;4;6;8]
```

We can add up the items in a list using the List.sum function:

```
let sum items =
    items |> List.sum

let mySum = sum myList // 45
```

Other aggregation functions are as easy to use but we are not going to look at them here.

Sometimes we want to perform an operation on each item in a list. If we want to return the new list, we use the List.map function:

```
let triple items =
    items
    |> List.map (fun x -> x * 3)

let myTriples = triple myList
```

If we don't want to return a new list, we use the List.iter function:

```
let print items =
    items
    |> List.iter (fun x -> (printfn "My value is %i" x))

print myList |> ignore
```

Let's take a look at a more complicated example using List.map that changes the structure of the output list. We will use a list of tuples (int * decimal) which might represent quantity and unit price.

```
let items = [(1,0.25M);(5,0.25M);(1,2.25M);(1,125M);(7,10.9M)]
```

To calculate the total price of the items, we can use the List.map function to convert an (int * decimal) list to decimal list and then sum the items:

```
let sum items =
    items
    |> List.map (fun (q, p) -> decimal q * p)
    |> List.sum
```

Note the explicit conversion of the integer to a decimal. F# is strict about types in calculations and doesn't support implicit conversion. Notice how we can pattern match in the lambda (fun (q, p) -> decimal q * p) to deconstruct the tuple to gain access to the contained values.

In this particular case, there is an easier way to do the calculation in one step with another of the List module functions:

```
let sum items =
    items
    |> List.sumBy (fun (q, p) -> decimal q * p)
```

Folding

A very powerful functional concept that we can use to do similar aggregation tasks (and lots more that we won't cover) is the List.fold function:

```
let getTotal items =
    items
    |> List.fold (fun acc (q, p) -> acc + decimal q * p) 0M

let total = getTotal items
```

The lambda function uses an accumulator and the deconstructed tuple and simply adds the intermediate calculation to the accumulator. The 0M parameter is the initial value of the accumulator. If we were folding using multiplication, the initial value would probably have been 1M.

An alternative style is to use another of the forward-pipe operators, (||>). This version supports a pair tupled of inputs:

```
let getTotal items =
    (0M, items) ||> List.fold (fun acc (q, p) -> acc + decimal q * p)

let total = getTotal items
```

There are situations where these additional operators can be useful but in simple case like this, I prefer the previous version.

Grouping Data and Uniqueness

If we wanted to get the unique numbers from a list, we can do it in many ways. Firstly we will use List.groupBy which will return a tuple for each distinct value:

```
let myList = [1;2;3;4;5;7;6;5;4;3]

// int list -> (int * int list) list
let gbResult = myList |> List.groupBy (fun x -> x)

// gbResult = [1,[1];2,[2];3,[3;3];4,[4;4];5,[5;5];6,[6];7,[7]]
```

To get the list of unique items from the result list, we can use the List.map function:

```
let unique items =
    items
    |> List.groupBy id
    |> List.map (fun (i, _) -> i)

let unResult = unique myList // [1;2;3;4;5;6;7]
```

The anonymous function (fun x -> x) can be replaced by *id*.

Using the List.groupBy function is nice but there is a function called List.distinct that will do exactly what we want:

```
let distinct = myList |> List.distinct
```

There is a built-in collection type called Set that will do this as well:

```
let uniqueSet items =
    items
    |> Set.ofList

let setResult = uniqueSet myList // [1;2;3;4;5;6;7]
```

Most of the collection types have a way of converting from and to each other.

The primary difference between the approaches is that using Set will sort the results if it can.

We will now put our new knowledge to good use.

Practical Example

Follow the instructions from the Appendix for creating a solution that we used in the setup of last chapter. The only difference is that you should create a classlib project rather than a console project.

In this example code we are going to manage an order with an immutable list of items. The functionality we need to add is:

- Add an item
- Increase quantity of an item
- Remove an item
- Reduce quantity of an item
- Clear all of the items

Create a new file in MyProject called Orders.fs. You can remove the Library.fs file. Add an OrderTests.fs file to MyProjectTests and add the namespace:

```
namespace MyProject.Orders
```

Add the following record types to Orders.fs:

```
type Item = {
    ProductId : int
    Quantity : int
}

type Order = {
    Id : int
    Items : Item list
}
```

Create a module called Domain and put it after the Order type definition:

```
module Domain =
```

We need to create a function to add an item to the order. This function needs to cater for products that exist in the order as well as those that don't. Let's think about how we can do this in pseudocode:

```
let addItem item order =

// Append item to order
// Count how many of each product we have
// New product = 1 entry, existing product = 2 entries
// Sort items in productid order to make equality simpler
// Update order with new items
```

Stage one is 'Append item to order':

```
// Append item to order
let addItem item order =
    let items = item::order.Items
    { order with Items = items }
```

Let's create a couple of helpers bindings and some simple asserts to help us test the function in FSI:

```
let order = { Id = 1; Items = [ { ProductId = 1; Quantity = 1 } ] }
let newItemExistingProduct = { ProductId = 1; Quantity = 1 }
let newItemNewProduct = { ProductId = 2; Quantity = 2 }

addItem newItemNewProduct order =
    { Id = 1; Items = [ { ProductId = 2; Quantity = 2 };{ ProductId = 1; Quantity = \
1 } ] }
addItem newItemExistingProduct order =
    { Id = 1; Items = [ { ProductId = 1; Quantity = 2 } ] }
```

Run this in FSI and do it for every change that we make. Look carefully at the output because it will point you to where you need to go next. If either of the asserts is false, run the code to the left of the equals sign on its own to see what it is returning.

Next we need to group the items per product using List.groupBy:

```
// Count how many of each product we have
let addItem item order =
    let items =
        item :: order.Items
        |> List.groupBy (fun i -> i.ProductId) // (int * Item list) list
    { order with Items = items }
```

This code won't compile as List.groupBy returns a tuple and this is not what order.Items is expecting. To view the result of the function, highlight the code in the function except the last line and run in FSI. Now run 'addItem newItemExistingProduct order' in FSI and look at the result.

We need to complete the function so that it returns the expected type:

```
// Fix the Items list
let addItem item order =
    let items =
        item::order.Items
        |> List.groupBy (fun i -> i.ProductId) // (int * Item list) list
        |> List.map (fun (id, items) ->
            { ProductId = id; Quantity = items |> List.sumBy (fun i -> i.Quantity) })
        |> List.sortBy (fun i -> i.ProductId)
    { order with Items = items }
```

Remove the helper bindings we have used for testing as we are going to create the following tests in a new OrderTests.fs file in MyProjectTests:

```
namespace OrderTests

open MyProject
open MyProject.Domain
open Xunit
open FsUnit

module ``Add item to order`` =

    [<Fact>]
    let ``product does not exist in empty order`` () =
        let myEmptyOrder = { Id = 1; Items = [] }
        let expected = { Id = 1; Items = [ { ProductId = 1; Quantity = 3 } ] }
        let actual = myEmptyOrder |> addItem { ProductId = 1; Quantity = 3 }
        actual |> should equal expected

    [<Fact>]
    let ``product does not exist in order`` () =
        let myOrder = { Id = 1; Items = [ { ProductId = 1; Quantity = 1 } ] }
        let expected = { Id = 1; Items = [ { ProductId = 1; Quantity = 1 };{ Product\
Id = 2; Quantity = 5 } ] }
        let actual = myOrder |> addItem { ProductId = 2; Quantity = 5 }
        actual |> should equal expected

    [<Fact>]
    let ``product exists in order`` () =
        let myOrder = { Id = 1; Items = [ { ProductId = 1; Quantity = 1 } ] }
        let expected = { Id = 1; Items = [ { ProductId = 1; Quantity = 4 } ] }
        let actual = myOrder |> addItem { ProductId = 1; Quantity = 3 }
        actual |> should equal expected
```

We have created three tests that cover the main cases when adding an item to an existing order.

Run the tests by running ‘dotnet test’ in the tests terminal. They should all pass.

We can easily add multiple items to an order by changing the cons operator (::) which adds a single item to the list into the concat operator (@) which joins two lists together:

```
// Item list -> Order -> Order
let addItems newItems order =
    let items =
        newItems @ order.Items
        |> List.groupBy (fun i -> i.ProductId)
        |> List.map (fun (id, items) ->
            { ProductId = id; Quantity = items |> List.sumBy (fun i -> i.Quantity) })
        |> List.sortBy (fun i -> i.ProductId)
    { order with Items = items }
```

We should add some tests to OrderTests.fs for the addItems function:

```
module ``add multiple items to an order`` =

    [<Fact>]
    let ``new products added to empty order`` () =
        let myEmptyOrder = { Id = 1; Items = [] }
        let expected = { Id = 1; Items = [ { ProductId = 1; Quantity = 1 }; { Produc\
tId = 2; Quantity = 5 } ] }
        let actual = myEmptyOrder |> addItems [ { ProductId = 1; Quantity = 1 }; { P\
roductId = 2; Quantity = 5 } ]
        actual |> should equal expected

    [<Fact>]
    let ``new products and updated existing to order`` () =
        let myOrder = { Id = 1; Items = [ { ProductId = 1; Quantity = 1 } ] }
        let expected = { Id = 1; Items = [ { ProductId = 1; Quantity = 2 }; { Produc\
tId = 2; Quantity = 5 } ] }
        let actual = myOrder |> addItems [ { ProductId = 1; Quantity = 1 }; { Produc\
tId = 2; Quantity = 5 } ]
        actual |> should equal expected
```

Run the tests to confirm your new code works.

Let’s extract the common functionality in addItem and addItems into a new function:

```
// Item list -> Item list
let recalculate items =
    items
    |> List.groupBy (fun i -> i.ProductId)
    |> List.map (fun (id, items) ->
        { ProductId = id; Quantity = items |> List.sumBy (fun i -> i.Quantity) })

let addItem item order =
    let items =
        item :: order.Items
        |> recalculate
        |> List.sortBy (fun i -> i.ProductId)
    { order with Items = items }

let addItems items order =
    let items =
        items @ order.Items
        |> recalculate
        |> List.sortBy (fun i -> i.ProductId)
    { order with Items = items }
```

Run the tests to confirm your changes have worked.

Removing a product can be easily achieved by filtering out the unwanted item by the ProductId:

```
let removeProduct productId order =
    let items =
        order.Items
        |> List.filter (fun x -> x.ProductId <> productId)
        |> List.sortBy (fun i -> i.ProductId)
    { order with Items = items }
```

Again we write some tests to verify our new function works as expected:

```
module ``Removing a product`` =

    [<Fact>]
    let ``remove all items of existing productid`` () =
        let myEmptyOrder = { Id = 1; Items = [ { ProductId = 1; Quantity = 1 } ] }
        let expected = { Id = 1; Items = [] }
        let actual = myEmptyOrder |> removeProduct 1
        actual |> should equal expected
    [<Fact>]
    let ``do nothing for non-existant productid`` () =
        let myOrder = { Id = 2; Items = [ { ProductId = 1; Quantity = 1 } ] }
        let expected = { Id = 2; Items = [ { ProductId = 1; Quantity = 1 } ] }
        let actual = myOrder |> removeProduct 2
        actual |> should equal expected
```

Reducing an item quantity is slightly more complex. We reduce the quantity of the item by the specified number, recalculate the items and then filter out any items with a quantity <= 0:

```
let reduceItem productId quantity order =
    let items =
        { ProductId = productId; Quantity = -quantity } :: order.Items
        |> recalculate
        |> List.filter (fun x -> x.Quantity > 0)
        |> List.sortBy (fun i -> i.ProductId)
    { order with Items = items }
```

Again we write some tests to verify our new function works as expected:

```
module ``Reduce item quantity`` =

    [<Fact>]
    let reduceSomeExistingItemAssert () =
        let myOrder = { Id = 1; Items = [ { ProductId = 1; Quantity = 5 } ] }
        let expected = { Id = 1; Items = [ { ProductId = 1; Quantity = 2 } ] }
        let actual = myOrder |> reduceItem 1 3
        actual |> should equal expected

    [<Fact>]
    let reduceAllExistingItemAssert () =
        let myOrder = { Id = 2; Items = [ { ProductId = 1; Quantity = 5 } ] }
        let expected = { Id = 2; Items = [] }
        let actual = myOrder |> reduceItem 1 5
        actual |> should equal expected

    [<Fact>]
    let reduceNonexistantItemAssert () =
        let myOrder = { Id = 3; Items = [ { ProductId = 1; Quantity = 1 } ] }
        let expected = { Id = 3; Items = [ { ProductId = 1; Quantity = 1 } ] }
        let actual = myOrder |> reduceItem 2 5
        actual |> should equal expected
    [<Fact>]
    let reduceNonexistantItemEmptyOrderAssert () =
        let myEmptyOrder = { Id = 4; Items = [] }
        let expected = { Id = 4; Items = [] }
        let actual = myEmptyOrder |> reduceItem 2 5
        actual |> should equal expected
```

Clearing all of the items is really simple. We just set Items to be an empty list:

```
let clearItems order =
    { order with Items = [] }
```

Write some tests to verify our new function works as expected:

```
module ``Empty an order of all items`` =

    [<Fact>]
    let ``order with existing items`` () =
        let myOrder = { Id = 1; Items = [ { ProductId = 1; Quantity = 1 } ] }
        let expected = { Id = 1; Items = [] }
        let actual = myOrder |> clearItems
        actual |> should equal expected

    [<Fact>]
    let ``empty order unchanged`` () =
        let myEmptyOrder = { Id = 2; Items = [] }
        let expected = { Id = 2; Items = [] }
        let actual = myEmptyOrder |> clearItems
        actual |> should equal expected
```

We have only scratched the surface of what is possible with collections in F#. For more details on the List module, have a look at the F# Docs[19].

Summary

In this chapter, we have looked at some of the most useful functions on the List module and we have seen that it is possible to use immutable data structures to provide important business functionality.

In the next chapter, we will look at how to handle a stream of data from a CSV source file.

[19]https://fsharp.github.io/fsharp-core-docs/reference/fsharp-collections-listmodule.html

6 - Reading Data From a File

In this chapter, we will introduce the basics of reading and parsing external data using sequences and the Seq module and learn how we can isolate code that talks to external services to make our codebase as testable as possible.

Setting Up

In a new folder for this chapter, create a console application by typing the following into a Terminal window:

```
dotnet new console -lang F#
```

Using the Explorer view, create a folder called 'resources' in the code folder for this chapter and then create a new file called 'customers.csv'. Copy the following data into the new file:

```
CustomerId|Email|Eligible|Registered|DateRegistered|Discount
John|john@test.com|1|1|2015-01-23|0.1
Mary|mary@test.com|1|1|2018-12-12|0.1
Richard|richard@nottest.com|0|1|2016-03-23|0.0
Sarah||0|0||
```

Remove all the code from Program.fs except the following:

```
open System

[<EntryPoint>]
let main argv =
    0
```

Loading Data

We are going to use the .NET System.IO classes to open the customers.csv file:

```
open System.IO
```

To load the data, we need to create a function that takes a path as a string, reads the file contents, and returns a collection of lines as strings from the file. Let's start with reading from a file:

```
let readFile path = // string -> seq<string>
    seq {
        use reader = new StreamReader(File.OpenRead(path))
        while not reader.EndOfStream do
            reader.ReadLine()
    }
```

There are a few new things in this simple function!

The seq {..} type is called a Sequence expression. The code inside the curly brackets will create a sequence of strings. We could have done the same with list and array but most of the System.IO methods output IEnumerable<'T> and seq {..} is the F# equivalent to IEnumerable<'T>.

The StreamReader class implements the IDisposable<'T> interface. Just like in C#, we need to make sure the code calls the Dispose() method when it has finished using it. To do this, we use the 'use' keyword. The indenting defines the scope of the disposable, in this case the Dispose() method will be called when the sequence has completed being generated. It is convention to use the 'new' keyword as well on instances that implement IDisposable<'T>.

Now we need to write some code in the main function to call our readFile function and output the data to the ouput window;

```
[<EntryPoint>]
let main argv =
    let path = Path.Combine(__SOURCE_DIRECTORY__, "resources", "customers.csv")
    let data = readFile path
    data |> Seq.iter (fun x -> printfn "%s" x)
    0
```

You must leave the '0' at the end of the main function.

We are using a built-in constant, __SOURCE_DIRECTORY__, to determine the current source code directory.

The Seq module has a wide range of functions available, similar to List and Array. The Seq.iter function will iterate over the sequence, perform an action, and return unit.

The code in Program.fs should now look like this:

```
open System
open System.IO

// string -> seq<string>
let readFile path =
    seq {
        use reader = new StreamReader(File.OpenRead(path))
        while not reader.EndOfStream do
            reader.ReadLine()
    }

[<EntryPoint>]
let main argv =
    let path = Path.Combine(__SOURCE_DIRECTORY__, "resources", "customers.csv")
    let data = readFile path
    data |> Seq.iter (fun x -> printfn "%s" x)
    0
```

Run the code by typing ‘dotnet run’ in the Terminal.

To handle potential errors from loading a file, we are going to add some error handling to the readFile function:

```
// string -> Result<seq<string>,exn>
let readFile path =
    try
        seq {
            use reader = new StreamReader(File.OpenRead(path))
            while not reader.EndOfStream do
                reader.ReadLine()
        }
        |> Ok
    with
    | ex -> Error ex
```

To handle the change in the signature of the readFile function, we will introduce a new function:

```
let import path =
    match path |> readFile with
    | Ok data -> data |> Seq.iter (fun x -> printfn "%A" x)
    | Error ex -> printfn "Error: %A" ex.Message
```

Replace the code in the main function with:

```
[<EntryPoint>]
let main argv =
    let path = Path.Combine(__SOURCE_DIRECTORY__, "resources", "customers.csv")
    import path
    0
```

We should simplify this code while we're here:

```
[<EntryPoint>]
let main argv =
    Path.Combine(__SOURCE_DIRECTORY__, "resources", "customers.csv")
    |> import
    0
```

Run the program to check it still works by typing the following in the Terminal:

```
dotnet run
```

Parsing Data

Now that we can load the data, we should try to convert each line to an F# record:

```
type Customer = {
    CustomerId : string
    Email : string
    IsEligible : string
    IsRegistered : string
    DateRegistered : string
    Discount : string
}
```

It is convention, and good practice, to put types definitions above let bindings in a code file.

Create a function that takes a seq<string> as input, parses the input data, and returns a seq<Customer> as output:

```
// seq<string> -> seq<Customer>
let parse (data:string seq) =
    data
    |> Seq.skip 1 // Ignore the header row
    |> Seq.map (fun line ->
        match line.Split('|') with
        | [| customerId; email; eligible; registered; dateRegistered; discount |] ->
            Some {
                CustomerId = customerId
                Email = email
                IsEligible = eligible
                IsRegistered = registered
                DateRegistered = dateRegistered
                Discount = discount
            }
        | _ -> None
    )
    |> Seq.choose id // Ignore None and unwrap Some
```

There are some new features in this function:

The Seq.skip function ignores a number of lines. In this case, the first item in the sequence is a header row and not a Customer, so it should be ignored.

The Split function creates an array of strings. We then pattern match the array and get the data which we then use to populate a Customer. If you aren't interested in all of the data, you can use the wildcard (_) for those parts. We have now met the three primary collection types in F#: List ([..]), Seq (seq {..}) and Array ([|..|]).

The Seq.choose function will ignore any item in the sequence that is None and will unwrap the Some items to return a sequence of Customers. The *id* keyword is a built-in value that is equivalent to (fun x -> x).

We need to modify the import function to use the new parse function:

```
let import path =
    match path |> readFile with
    | Ok data -> data |> parse |> Seq.iter (fun x -> printfn "%A" x)
    | Error ex -> printfn "Error: %A" ex.Message
```

The next stage is to extract the code from the Seq.map in the parse function into its own function:

```
// string -> Customer option
let parseLine (line:string) : Customer option =
    match line.Split('|') with
    | [| customerId; email; eligible; registered; dateRegistered; discount |] ->
        Some {
            CustomerId = customerId
            Email = email
            IsEligible = eligible
            IsRegistered = registered
            DateRegistered = dateRegistered
            Discount = discount
        }
    | _ -> None
```

Modify the parse function to use the new parseLine function:

```
let parse (data:string seq) =
    data
    |> Seq.skip 1
    |> Seq.map (fun x -> parseLine x)
    |> Seq.choose id
```

We can simplify this function by removing the lambda:

```
let parse (data:string seq) =
    data
    |> Seq.skip 1
    |> Seq.map parseLine
    |> Seq.choose id
```

Testing the Code

Whilst we have improved the code a lot, it is difficult to test without having to load a file. The signature of the readFile function is (string -> Result<seq<string>>,exn>) which means that we could easily convert it to use a webservice rather than a path to a file on disk.

To make this testable and extensible, we can make import a higher order function by taking a function as a parameter with the same signature as readFile as a parameter. We should set the name of the input parameter to dataReader to reflect the possibility of different sources of data than a file:

```
let output data =
    data
    |> Seq.iter (fun x -> printfn "%A" x)

let import (dataReader:string -> Result<string seq,exn>) path =
    match path |> dataReader with
    | Ok data -> data |> parse |> output
    | Error ex -> printfn "Error: %A" ex.Message
```

We can now pass any function with this signature (string -> Result<string seq,exn>) into the import function.

This signature is quite simple but they can get quite complex. We can create a Function type with the same signature and use that in the parameter instead:

```
type DataReader = string -> Result<string seq,exn>
```

Replace the function signature in import with it;

```
let import (dataReader:DataReader) path =
    match path |> dataReader with
    | Ok data -> data |> parse |> output
    | Error ex -> printfn "Error: %A" ex.Message
```

We can use it like an Interface in the readFile function but it does mean modifying our code a little to use an alternate function style:

```
let readFile : DataReader =
    fun path ->
        try
            seq {
                use reader = new StreamReader(File.OpenRead(path))
                while not reader.EndOfStream do
                    reader.ReadLine() }
            |> Ok
        with
        | ex -> Error ex
```

We need to make a small change to our call in main to tell it to use the readFile function as we have changed the signature of the import function:

```
[<EntryPoint>]
let main argv =
    Path.Combine(__SOURCE_DIRECTORY__, "resources", "customers.csv")
    |> import readFile
    0
```

If we use import with `readFile` regularly, we can use partial application to create a new function that does that for us:

```
let importWithFileReader = import readFile
```

To use it we would simply call:

```
importWithFileReader Path.Combine(__SOURCE_DIRECTORY__, "resources", "customers.csv")
```

The payoff for the work we have done using Higher Order functions and Function types is that we can easily pass in a fake function for testing:

```
let fakeDataReader : DataReader =
    fun _ ->
        seq {
            "CustomerId|Email|Eligible|Registered|DateRegistered|Discount"
            "John|john@test.com|1|1|2015-01-23|0.1"
            "Mary|mary@test.com|1|1|2018-12-12|0.1"
            "Richard|richard@nottest.com|0|1|2016-03-23|0.0"
            "Sarah||0|0||"
        }
        |> Ok

import fakeDataReader "_"
```

You can use any function that satisfies the DataReader function signature.

Final Code

What we have ended up with is the following;

```
open System.IO

type Customer = {
    CustomerId : string
    Email : string
    IsEligible : string
    IsRegistered : string
    DateRegistered : string
    Discount : string
}

type DataReader = string -> Result<string seq,exn>

let readFile : DataReader =
    fun path ->
        try
            seq {
                use reader = new StreamReader(File.OpenRead(path))
                while not reader.EndOfStream do
                    reader.ReadLine()
            }
            |> Ok
        with
        | ex -> Error ex

let parseLine (line:string) : Customer option =
    match line.Split('|') with
    | [| customerId; email; eligible; registered; dateRegistered; discount |] ->
        Some {
            CustomerId = customerId
            Email = email
            IsEligible = eligible
            IsRegistered = registered
            DateRegistered = dateRegistered
            Discount = discount
        }
    | _ -> None

let parse (data:string seq) =
    data
    |> Seq.skip 1
    |> Seq.map parseLine
    |> Seq.choose id
```

```
let output data =
    data
    |> Seq.iter (fun x -> printfn "%A" x)

let import (dataReader:DataReader) path =
    match path |> dataReader with
    | Ok data -> data |> parse |> output
    | Error ex -> printfn "Error: %A" ex.Message

[<EntryPoint>]
let main argv =
    Path.Combine(__SOURCE_DIRECTORY__, "resources", "customers.csv")
    |> import readFile
    0
```

In a later chapter, we will extend this code by adding data validation.

Summary

In this chapter, we have looked at how we can import data using some of the most useful functions on the Seq module, sequence expressions and function types. We have also seen how we can use higher order functions to allow us to easily extend functions to take different functions at runtime and for tests.

In the next chapter, we will look at another exciting F# feature, Active Patterns, to help make our pattern matching more readable.

7 - Active Patterns

In this chapter we will be extending our knowledge of pattern patching by looking at how we can write our own matchers with Active Patterns. There are a number of different Active Patterns types available and we will look at most of them in this chapter.

Setting Up

Create a folder for the code from this chapter.

All of the code in this chapter can be written in an F# script file (.fsx) and run using F# Interactive.

Partial Active Patterns

One area where Partial Active Patterns are really helpful is in validation and parsing. This is an example of parsing a string to a DateTime without using an active pattern:

```
open System

// string -> DateTime option
let parse (input:string) =
    match DateTime.TryParse(input) with
    | true, value -> Some value
    | false, _ -> None

let isDate = parse "2019-12-20" // Some 2019-12-20 00:00:00
let isNotDate = parse "Hello" // None
```

It works but we can't use it directly in a pattern match, only in a guard clause. Let's create a partial active pattern to handle the DateTime parsing for us:

```
open System

// string -> DateTime option
let (|ValidDate|_|) (input:string) =
    let success, value = DateTime.TryParse(input)
    if success then Some value else None
```

Anytime that you see the 'banana clips' (|…|), you are looking at an active pattern.

Now we can plug it into our parse function:

```
let parse input =
    match input with
    | ValidDate dt -> printfn "%A" dt
    | _ -> printfn "'%s{input}' is not a valid date"

let isDate = parse "2019-12-20" // 2019-12-20 00:00:00
let isNotDate = parse "Hello" // 'Hello' is not a valid date
```

There's more code but it is much more readable. In this case, the active pattern returns the parsed value. If you didn't care about the value being returned, you can return unit Some () instead of Some value.

Parameterized Partial Active Patterns

We are going to investigate how an old Interview favourite - FizzBuzz - can be implemented using an active pattern. Let's start with the canonical solution:

```
let calculate i =
    if i % 3 = 0 && i % 5 = 0 then "FizzBuzz"
    elif i % 3 = 0 then "Fizz"
    elif i % 5 = 0 then "Buzz"
    else i |> string

[1..15] |> List.map calculate
```

It works but can we do better with Pattern Matching? How about this?

```
let calculate i =
    match (i % 3, i % 5) with
    | (0, 0) -> "FizzBuzz"
    | (0, _) -> "Fizz"
    | (_, 0) -> "Buzz"
    | _ -> i |> string
```

or this?

```
let calculate i =
    match (i % 3 = 0, i % 5 = 0) with
    | (true, true) -> "FizzBuzz"
    | (true, _) -> "Fizz"
    | (_, true) -> "Buzz"
    | _ -> i |> string
```

Neither of these is any more readable than the original. How about if we could do something like this?

```
let calculate i =
    match i with
    | IsDivisibleBy 3 & IsDivisibleBy 5 -> "FizzBuzz"
    | IsDivisibleBy 3 -> "Fizz"
    | IsDivisibleBy 5 -> "Buzz"
    | _ -> i |> string
```

Note the single '&' to apply both parts of the pattern match in the calculate function.

We can use a Parameterized Partial Active Pattern to do this:

```
// int -> int -> unit option
let (|IsDivisibleBy|_|) divisor n  =
    if n % divisor = 0 then Some () else None
```

The parameterized name comes from the fact that we can supply additional parameters. The value being tested must always be the last parameter. In this case we include the divisor parameter as well as the number we are checking (n).

This is much nicer but what happens if we need to add '7 -> Bazz' or even more options into the mix? Look at the code now!

```
let calculate i =
    match i with
    | IsDivisibleBy 3 & IsDivisibleBy 5 & IsDivisibleBy 7 -> "FizzBuzzBazz"
    | IsDivisibleBy 3 & IsDivisibleBy 5 -> "FizzBuzz"
    | IsDivisibleBy 3 & IsDivisibleBy 7 -> "FizzBazz"
    | IsDivisibleBy 5 & IsDivisibleBy 7 -> "BuzzBazz"
    | IsDivisibleBy 3 -> "Fizz"
    | IsDivisibleBy 5 -> "Buzz"
    | IsDivisibleBy 7 -> "Bazz"
    | _ -> i |> string
```

Maybe not such a good idea? How confident would you be that you had covered all of the permutations? Maybe a different approach would yield a better result?

```
let calculate n =
    [(3, "Fizz"); (5, "Buzz"); (7, "Bazz")]
    |> List.map (fun (divisor, result) -> if n % divisor = 0 then result else "")
    |> List.reduce (+)
    |> fun input -> if input = "" then string n else input

[1..15] |> List.map calculate
```

The List.reduce function will concatenate all of the strings generated by the List.map function. The last line is to cater for numbers that are not divisible by any of the numbers in the initial list.

It is less readable than the original but it's much less prone to errors covering all of the possible permutations. You could even pass the mappings in as a parameter:

```
let calculate mapping n =
    mapping
    |> List.map (fun (divisor, result) -> if n % divisor = 0 then result else "")
    |> List.reduce (+)
    |> fun input -> if input = "" then string n else input

[1..15] |> List.map (calculate [(3, "Fizz"); (5, "Buzz")])
```

This is the type of function that F# devs write all the time.

Back to active patterns! Let's have a look at another favourite interview question - Leap Years. The basic F# function looks like this;

```
let isLeapYear year =
    year % 400 = 0 || (year % 4 = 0 && year % 100 <> 0)

[2000;2001;2020] |> List.map isLeapYear = [true;false;true]
```

But can parameterized partial active pattern help make it more readable? How about two of them?

```
let (|IsDivisibleBy|_|) divisor n  =
    if n % divisor = 0 then Some () else None

let (|NotDivisibleBy|_|) divisor n  =
    if n % divisor <> 0 then Some () else None

let isLeapYear year =
    match year with
    | IsDivisibleBy 400 -> true
    | IsDivisibleBy 4 & NotDivisibleBy 100 -> true
    | _ -> false
```

There's more code but you could easily argue that it is more readable. We could also do a similar thing to the original with helper functions rather than Active Patterns;

```
let isDivisibleBy divisor year =
    year % divisor = 0

let notDivisibleBy divisor year =
    not (year |> isDivisibleBy divisor)

let isLeapYear year =
    year |> isDivisibleBy 400 || (year |> isDivisibleBy 4 && year |> notDivisibleBy \
100)
```

You could also use a match expression with guard clauses:

```
let isLeapYear input =
    match input with
    | year when year |> isDivisibleBy 400 -> true
    | year when year |> isDivisibleBy 4 && year |> notDivisibleBy 100 -> true
    | _ -> false
```

All of these approaches are valid and it's down to preference which is best.

Multi-Case Active Patterns

Multi-Case Active Patterns are different from Partial Active Patterns in that they allow more than one choice and return the selected one rather than an option type. The maximum number of choices supported is currently seven although that may increase in a future version of F#.

I play, rather unsuccessfully, in an online Football (the sport where participants kick a spherical ball with their feet rather than throw an egg-shaped object with their hands) Score Predictor.

The rules are simple;

- 300 points for predicting the correct score (2-3 vs 2-3)
- 100 points for predicting the correct result (2-3 vs 0-2)
- 15 points per home goal & 20 points per away goal using the lower of the predicted and actual scores

We have some sample predictions, actual scores and points that we can use to validate the code we write;

```
(0, 0) (0, 0) = 400 // 300 + 100 + 0 * 15 + 0 * 20
(3, 2) (3, 2) = 485 // 300 + 100 = 3 * 15 + 2 * 20
(5, 1) (4, 3) = 180 // 100 + 4 * 15 + 1 * 20
(2, 1) (0, 7) = 20 // 0 * 15 + 1 * 20
(2, 2) (3, 3) = 170 //100 + 2 * 15 + 2 * 20
```

Firstly we create a simple tuple type to represent a score;

```
type Score = int * int
```

To determine if we have a correct score, we need to check that the prediction and the actual score are the same. Since F# uses structural equality rather than reference equality, this is trivial with a partial active pattern:

```
let (|CorrectScore|_|) (expected:Score, actual:Score) =
    if expected = actual then Some () else None
```

We don’t care about the actual scores, just that they are the same, so we can return unit.

We also need to determine what the result of a Score is. This can only one of three choices; Draw (Tie), Home win or Away win. We can easily represent this with a multi-case active pattern:

```
let (|Draw|HomeWin|AwayWin|) (score:Score) =
    match score with
    | (h, a) when h = a -> Draw
    | (h, a) when h > a -> HomeWin
    | _ -> AwayWin
```

Note that this active pattern returns one of the pattern choices rather than an option type. We can now create a new partial active pattern for determining if we have predicted the correct result:

```
let (|CorrectResult|_|) (expected:Score, actual:Score) =
    match (expected, actual) with
    | (Draw, Draw) -> Some ()
    | (HomeWin, HomeWin) -> Some ()
    | (AwayWin, AwayWin) -> Some ()
    | _ -> None
```

We have to provide the input data as a single structure, so we use a tuple of Scores.

Without the multi-case active pattern for the result of a score, we would have to write something like this:

```
let (|CorrectResult|_|) (expected:Score, actual:Score) =
    match (expected, actual) with
    | ((h, a), (h', a')) when h = a && h' = a' -> Some ()
    | ((h, a), (h', a')) when h > a && h' > a' -> Some ()
    | ((h, a), (h', a')) when h < a && h' < a' -> Some ()
    | _ -> None
```

I prefer the version using the multi-case active pattern.

Now we need to create a function to work out the points for the goals scored:

```
let goalsScore (expected:Score) (actual:Score) =
    let (h, a) = expected
    let (h', a') = actual
    let home = [ h; h' ] |> List.min
    let away = [ a; a' ] |> List.min
    (home * 15) + (away * 20)
```

We now have all of the parts to create our function to calculate the total points for each game:

```
let calculatePoints (expected:Score) (actual:Score) =
    let pointsForCorrectScore =
        match (expected, actual) with
        | CorrectScore -> 300
        | _ -> 0
    let pointsForCorrectResult =
        match (expected, actual) with
        | CorrectResult -> 100
        | _ -> 0
    let pointsForGoals = goalsScore expected actual
    pointsForCorrectScore + pointsForCorrectResult + pointsForGoals
```

Note how the tuple in the match expression matches the input required by the parameterized active patterns.

There is a bit of duplication/similarity that we will resolve later but firstly we should use our test data to validate our code using FSI:

```
let assertnoScoreDrawCorrect =
    calculatePoints (0, 0) (0, 0) = 400
let assertHomeWinExactMatch =
    calculatePoints (3, 2) (3, 2) = 485
let assertHomeWin =
    calculatePoints (5, 1) (4, 3) = 180
let assertIncorrect =
    calculatePoints (2, 1) (0, 7) = 20
let assertDraw =
    calculatePoints (2, 2) (3, 3) = 170
```

We can simplify the calculatePoints function by combining the pattern matching for CorrectScore and CorrectResult into a new function:

```
let resultScore (expected:Score) (actual:Score) =
    match (expected, actual) with
    | CorrectScore -> 400
    | CorrectResult -> 100
    | _ -> 0
```

Note that we had to return 400 from CorrectScore in this function as we are no longer able to add the CorrectResult points later. This allows us to simplify the calculatePoints function:

```
let calculatePoints (expected:Score) (actual:Score) =
    let pointsForResult = resultScore expected actual
    let pointsForGoals = goalScore expected actual
    pointsForResult + pointsForGoals
```

As the resultScore and goalScore functions have the same signature, we can use a higher order function to remove the duplication:

```
let calculatePoints (expected:Score) (actual:Score) =
    [ resultScore; goalScore ]
    |> List.sumBy (fun f -> f expected actual)
```

Yes, we can put functions into a List. List.sumBy is equivalent to List.map followed by List.sum.

There are other types of active pattern that we haven't met in this chapter; I recommend that you investigate the F# Documentation to make yourself familiar with them.

Summary

Active patterns are very useful and powerful features but they are not always the best approach. When used well, they improve readability.

In the next chapter we will be taking some of the features we've met in this chapter to add validation to the code we used in chapter 6.

8 - Functional Validation

In this post we are going to look at adding validation to a copy of the code we created in Chapter 6. We will use Active Patterns that we looked at in the last chapter and we will see how you can easily model domain errors. At the end of the chapter, we will have a quick look at one of the more interesting functional patterns which is perfect for the validation that we are doing.

Setting Up

Create a new folder for the code in this chapter.

Create a console application by typing the following into a terminal window:

```
dotnet new console -lang F#
```

Using the Explorer view, create a folder called 'resources' in the code folder for this chapter and then create a new file called customers.csv. Copy the following data into the new file:

```
CustomerId|Email|Eligible|Registered|DateRegistered|Discount
John|john@test.com|1|1|2015-01-23|0.1
Mary|mary@test.com|1|1|2018-12-12|0.1
Richard|richard@nottest.com|0|1|2016-03-23|0.0
Sarah||0|0||
```

Replace the code in Program.fs with the following:

```
open System
open System.IO

type Customer = {
    CustomerId : string
    Email : string
    IsEligible : string
    IsRegistered : string
    DateRegistered : string
    Discount : string
}
```

```
type DataReader = string -> Result<string seq,exn>

let readFile : DataReader =
    fun path ->
        try
            seq {
                use reader = new StreamReader(File.OpenRead(path))
                while not reader.EndOfStream do
                    reader.ReadLine()
            }
            |> Ok
        with
        | ex -> Error ex

let parseLine (line:string) : Customer option =
    match line.Split('|') with
    | [| customerId; email; eligible; registered; dateRegistered; discount |] ->
        Some {
            CustomerId = customerId
            Email = email
            IsEligible = eligible
            IsRegistered = registered
            DateRegistered = dateRegistered
            Discount = discount
        }
    | _ -> None

let parse (data:string seq) =
    data
    |> Seq.skip 1
    |> Seq.map parseLine
    |> Seq.choose id

let output data =
    data
    |> Seq.iter (fun x -> printfn "%A" x)

let import (dataReader:DataReader) path =
    match path |> dataReader with
    | Ok data -> data |> parse |> output
    | Error ex -> printfn "Error: %A" ex.Message

[<EntryPoint>]
```

```
let main argv =
    Path.Combine(__SOURCE_DIRECTORY__, "resources", "customers.csv")
    |> import readFile
    0
```

Run it using ‘dotnet run’ in the terminal.

Solving the Problem

The first thing we need to do is create a new record type that will store our validated data. Add this type below the Customer record type definition:

```
type ValidatedCustomer = {
    CustomerId : string
    Email : string option
    IsEligible : bool
    IsRegistered : bool
    DateRegistered : DateTime option
    Discount : decimal option
}
```

Have a look at the source data in the CSV file to understand why some of the parts are optional.

We need to add a new function to create a ValidatedCustomer:

```
let create customerId email isEligible isRegistered dateRegistered discount =
    {
        CustomerId = customerId
        Email = email
        IsEligible = isEligible
        IsRegistered = isRegistered
        DateRegistered = dateRegistered
        Discount = discount
    }
```

Now we need to think about how we handle validation errors. The obvious choices are using Option but then we lose the reason for the error or Result but that is going to make using our new create function difficult to use: We will use Result. The other thing we need to consider is what types of errors do we expect. The obvious ones are missing data (empty string) or invalid data (string to DateTime/decimal/boolean). We also want to return all of the errors, not just the first one, so we need a list of errors in the output. Lets create the error type as a discriminated union:

```
type ValidationError =
    | MissingData of name: string
    | InvalidData of name: string * value: string
```

For missing data, we only need the name of the item but for invalid data, we want the name and the value that failed.

You will need to add an import declaration at the top of the file:

```
open System.Text.RegularExpressions
```

Now we will create some helper functions using active patterns to handle parsing empty string, email regex and booleans etc:

```
let (|ParseRegex|_|) regex str =
    let m = Regex(regex).Match(str)
    if m.Success then Some (List.tail [ for x in m.Groups -> x.Value ])
    else None

let (|IsValidEmail|_|) input =
    match input with
    | ParseRegex ".*?@(.*)" [ _ ] -> Some input
    | _ -> None

let (|IsEmptyString|_|) (input:string) =
    if input.Trim() = "" then Some () else None

let (|IsDecimal|_|) (input:string) =
    let success, value = Decimal.TryParse input
    if success then Some value else None

let (|IsBoolean|_|) (input:string) =
    match input with
    | "1" -> Some true
    | "0" -> Some false
    | _ -> None
```

Now let's create our validate functions using our new active patterns and the new ValidationError discriminated union type:

```
// string -> Result<string, ValidationError>
let validateCustomerId customerId =
    if customerId <> "" then Ok customerId
    else Error (MissingData "CustomerId")

// string -> Result<string option, ValidationError>
let validateEmail email =
    if email <> "" then
        match email with
        | IsValidEmail _ -> Ok (Some email)
        | _ -> Error (InvalidData ("Email", email))
    else
        Ok None

// string -> Result<bool, ValidationError>
let validateIsEligible (isEligible:string) =
    match isEligible with
    | IsBoolean b -> Ok b
    | _ -> Error (InvalidData ("IsEligible", isEligible))

// string -> Result<bool, ValidationError>
let validateIsRegistered (isRegistered:string) =
    match isRegistered with
    | IsBoolean b -> Ok b
    | _ -> Error (InvalidData ("IsRegistered", isRegistered))

// string -> Result<DateTime option, ValidationError>
let validateDateRegistered (dateRegistered:string) =
    match dateRegistered with
    | IsEmptyString -> Ok None
    | _ ->
        let (success, value) = dateRegistered |> DateTime.TryParse
        if success then Ok (Some value)
        else Error (InvalidData ("DateRegistered", dateRegistered))

// string -> Result<decimal option, ValidationError>
let validateDiscount discount =
    match discount with
    | IsEmptyString -> Ok None
    | IsDecimal value -> Ok (Some value)
    | _ -> Error (InvalidData ("Discount", discount))
```

We now need to create a function that uses our validation functions on the customer properties and will create a ValidatedCustomer instance. Notice that I have added the expected return type to the new function:

```
let validate (input:Customer) : Result<ValidatedCustomer, ValidationError list> =
    let customerId = input.CustomerId |> validateCustomerId
    let email = input.Email |> validateEmail
    let isEligible = input.IsEligible |> validateIsEligible
    let isRegistered = input.IsRegistered |> validateIsRegistered
    let dateRegistered = input.DateRegistered |> validateDateRegistered
    let discount = input.Discount |> validateDiscount
    // This won't compile
    create customerId email isEligible isRegistered dateRegistered discount
```

We now have a problem: The create function isn't expecting Result types from the validation functions. With the skills and knowledge that we have gained so far, we can solve this.

Firstly, we create a couple of helper functions to extract Error and Ok data:

```
let getError input =
    match input with
    | Ok _ -> []
    | Error ex -> [ ex ]

let getValue input =
    match input with
    | Ok v -> v
    | _ -> failwith "Oops, you shouldn't have got here!"
```

Now we create a list of potential errors using a List Comprehension, concatenate them and then check to see if there are any. If there are no errors, we can safely call the create function:

```
let validate (input:Customer) : Result<TypedCustomer, ConversionError list> =
    let customerId = input.CustomerId |> validateCustomerId
    let email = input.Email |> validateEmail
    let isEligible = input.IsEligible |> validateIsEligible
    let isRegistered = input.IsRegistered |> validateIsRegistered
    let dateRegistered = input.DateRegistered |> validateDateRegistered
    let discount = input.Discount |> validateDiscount
    let errors =
        [
            customerId |> getError;
            email |> getError;
            isEligible |> getError;
            isRegistered |> getError;
            dateRegistered |> getError;
            discount |> getError
        ]
        |> List.concat
    match errors with
    | [] -> Ok (create (customerId |> getValue) (email |> getValue) (isEligible |> g\
etValue) (isRegistered |> getValue) (dateRegistered |> getValue) (discount|> getValu\
e))
    | _ -> Error errors
```

Finally, we need to plug the validation into the pipeline:

```
// seq<string> -> seq<Result<ValidatedCustomer, ValidationError list>>
let parse (data:string seq) =
    data
    |> Seq.skip 1
    |> Seq.map parseLine
    |> Seq.choose id
    |> Seq.map validate
```

If you run the code using 'dotnet run' in the terminal, you should get some validated customer data as output.

Add an extra row to the customers.csv file:

```
|||||
```

If you run the code again, the last item in the output to the terminal will be an error.

Where are we now?

This is the code we have ended up with:

```
open System
open System.IO
open System.Text.RegularExpressions

type Customer = {
    CustomerId : string
    Email : string
    IsEligible : string
    IsRegistered : string
    DateRegistered : string
    Discount : string
}

type ValidatedCustomer = {
    CustomerId : string
    Email : string option
    IsEligible : bool
    IsRegistered : bool
    DateRegistered : DateTime option
    Discount : decimal option
}

type ValidationError =
| MissingData of name: string
| InvalidData of name: string * value: string

type FileReader = string -> Result<string seq, exn>

let readFile : FileReader =
    fun path ->
        try
            seq {
                use reader = new StreamReader(File.OpenRead(path))
                while not reader.EndOfStream do
                    yield reader.ReadLine()
            }
            |> Ok
        with
        | ex -> Error ex

let parseLine (line:string) : Customer option =
    match line.Split('|') with
    | [| customerId; email; eligible; registered; dateRegistered; discount |] ->
```

```
        Some {
            CustomerId = customerId
            Email = email
            IsEligible = eligible
            IsRegistered = registered
            DateRegistered = dateRegistered
            Discount = discount
        }
    | _ -> None

let (|ParseRegex|_|) regex str =
    let m = Regex(regex).Match(str)
    if m.Success then Some (List.tail [ for x in m.Groups -> x.Value ])
    else None

let (|IsValidEmail|_|) input =
    match input with
    | ParseRegex ".*?@(.*)" [ _ ] -> Some input
    | _ -> None

let (|IsEmptyString|_|) (input:string) =
    if input.Trim() = "" then Some () else None

let (|IsDecimal|_|) (input:string) =
    let success, value = Decimal.TryParse input
    if success then Some value else None

let (|IsBoolean|_|) (input:string) =
    match input with
    | "1" -> Some true
    | "0" -> Some false
    | _ -> None

// string -> Result<string, ValidationError>
let validateCustomerId customerId =
    if customerId <> "" then Ok customerId
    else Error (MissingData "CustomerId")

// string -> Result<string option, ValidationError>
let validateEmail email =
    if email <> "" then
        match email with
        | IsValidEmail _ -> Ok (Some email)
        | _ -> Error (InvalidData ("Email", email))
    else
        Ok None

// string -> Result<bool, ValidationError>
let validateIsEligible (isEligible:string) =
    match isEligible with
    | IsBoolean b -> Ok b
    | _ -> Error (InvalidData ("IsEligible", isEligible))

// string -> Result<bool, ValidationError>
let validateIsRegistered (isRegistered:string) =
    match isRegistered with
    | IsBoolean b -> Ok b
    | _ -> Error (InvalidData ("IsRegistered", isRegistered))

// string -> Result<DateTime option, ValidationError>
let validateDateRegistered (dateRegistered:string) =
    match dateRegistered with
    | IsEmptyString -> Ok None
    | _ ->
        let (success, value) = dateRegistered |> DateTime.TryParse
        if success then Ok (Some value)
        else Error (InvalidData ("DateRegistered", dateRegistered))

// string -> Result<decimal option, ValidationError>
let validateDiscount discount =
    match discount with
    | IsEmptyString -> Ok None
    | IsDecimal value -> Ok (Some value)
    | _ -> Error (InvalidData ("Discount", discount))

let getError input =
    match input with
    | Ok _ -> []
    | Error ex -> [ ex ]

let getValue input =
    match input with
    | Ok v -> v
    | _ -> failwith "Oops, you shouldn't have got here!"

let create customerId email isEligible isRegistered dateRegistered discount =
    {
        CustomerId = customerId
        Email = email
        IsEligible = isEligible
        IsRegistered = isRegistered
        DateRegistered = dateRegistered
        Discount = discount
    }

let validate (input:Customer) : Result<ValidatedCustomer, ValidationError list> =
    let customerId = input.CustomerId |> validateCustomerId
    let email = input.Email |> validateEmail
    let isEligible = input.IsEligible |> validateIsEligible
    let isRegistered = input.IsRegistered |> validateIsRegistered
    let dateRegistered = input.DateRegistered |> validateDateRegistered
    let discount = input.Discount |> validateDiscount
    let errors =
        [
            customerId |> getError;
            email |> getError;
            isEligible |> getError;
            isRegistered |> getError;
            dateRegistered |> getError;
            discount |> getError
        ]
        |> List.concat
    match errors with
    | [] -> Ok (create (customerId |> getValue) (email |> getValue) (isEligible |> g\
etValue) (isRegistered |> getValue) (dateRegistered |> getValue) (discount|> getValu\
e))
    | _ -> Error errors

let parse (data:string seq) =
    data
    |> Seq.skip 1
    |> Seq.map parseLine
    |> Seq.choose id
    |> Seq.map validate

let output data =
    data
    |> Seq.iter (fun x -> printfn "%A" x)
```

```
let import (fileReader:FileReader) path =
    match path |> fileReader with
    | Ok data -> data |> parse |> output
    | Error ex -> printfn "Error: %A" ex

[<EntryPoint>]
let main argv =
    Path.Combine(__SOURCE_DIRECTORY__, "resources", "customers.csv")
    |> import readFile
    0
```

This code works well but there is a more idiomatic way of handling this type of problem: Applicatives.

Functional validation the F# way

We are going to use the **Computation Expression** support for applicatives in F# 5. We are not going to deep dive into computation expressions in this chapter but will do so in Chapter 12. It is enough at this stage to know how they are used.

> If you want to see an explaination of using applicatives for validation using the idiomatic style prior to F# 5, have a look at the post on Functional Validation in F# Using Applicatives[20] that I wrote for the 2019 F# Advent Calendar[21]. It's well worth understanding how it works under the covers as it builds on many of the features we have encountered so far.

The first thing that we need to do is to convert our errors into ValidationError lists. We can use the built-in Result.mapError function to help us do this for each validated item:

```
let validate (input:Customer) : Result<ValidatedCustomer, ValidationError list> =
    let customerId =
        input.CustomerId
        |> validateCustomerId
        |> Result.mapError (fun ex -> [ ex ])
    let email =
        input.Email
        |> validateEmail
        |> Result.mapError (fun ex -> [ ex ])
    let isEligible =
```

[20]https://trustbit.tech/blog/2019/12/09/functional-validation-in-f-using-applicatives
[21]https://sergeytihon.com/2019/11/05/f-advent-calendar-in-english-2019/

```
        input.IsEligible
        |> validateIsEligible
        |> Result.mapError (fun ex -> [ ex ])
    let isRegistered =
        input.IsRegistered
        |> validateIsRegistered
        |> Result.mapError (fun ex -> [ ex ])
    let dateRegistered =
        input.DateRegistered
        |> validateDateRegistered
        |> Result.mapError (fun ex -> [ ex ])
    let discount =
        input.Discount
        |> validateDiscount
        |> Result.mapError (fun ex -> [ ex ])
    // Compile problem
    create customerId email isEligible isRegistered dateRegistered discount
```

We are going to use some code from a NuGet package that we need to download. Enter the following in the terminal:

```
dotnet add package FsToolkit.ErrorHandling
```

After the package has downloaded, add the following import declaration:

```
open FsToolkit.ErrorHandling.ValidationCE
```

To finish, we need to plug in the validation computation expression:

```
let validate (input:Customer) : Result<ValidatedCustomer, ValidationError list> =
    validation {
        let! customerId =
            input.CustomerId
            |> validateCustomerId
            |> Result.mapError (fun ex -> [ ex ])
        and! email =
            input.Email
            |> validateEmail
            |> Result.mapError (fun ex -> [ ex ])
        and! isEligible =
            input.IsEligible
            |> validateIsEligible
            |> Result.mapError (fun ex -> [ ex ])
        and! isRegistered =
            input.IsRegistered
            |> validateIsRegistered
            |> Result.mapError (fun ex -> [ ex ])
        and! dateRegistered =
            input.DateRegistered
            |> validateDateRegistered
            |> Result.mapError (fun ex -> [ ex ])
        and! discount =
            input.Discount
            |> validateDiscount
            |> Result.mapError (fun ex -> [ ex ])
        return create customerId email isEligible isRegistered dateRegistered discou\
nt
    }
```

The key things to notice are the bangs (!) and the new to F# in version 5, *and* keyword.

The bang unwraps the effected value from the Ok track of the validated item. Move your mouse over customerId and the tooltip will tell you it is a string. Remove the bang and the tooltip will inform you that customerId is an option<string>.

If we had used *let* rather than *and*, the code would have only returned the first error found and then would not have called anything else as it would be on the Error track. The *and* forces the code to run all of the validation. The final function is only called if there are no errors. In effect, it ends up doing exactly what we did initially but much more elegantly!

Computation expressions are the only place in F# where the *return* keyword is required.

Don't worry if you don't fully understand this code as we will be covering Computation Expressions in detail in chapter 12 and when we start using Giraffe in chapter 13.

Check that everything is working by running the code from the terminal:

```
dotnet run
```

Applicatives are very useful for a range of things, so it's an important pattern to know, even in the old style.

Summary

In this post we have looked at how we can add validation by using active patterns and how easy it is to add additional functionality into the data processing pipeline. We also looked at a more elegant solution than our original one to the validation problem using applicatives with the computation expression support in F# 5.

In the next chapter, we will look at improving the code from the first chapter by using more domain terminology.

9 - Single Case Discriminated Union

In this chapter, we are going to see how we can improve the readability of our code by increasing our usage of domain concepts and reducing our use of primitives.

Setting Up

We are going to improve the code we wrote in Chapter 1.

Create a new folder for the code in this chapter and open the new folder in VS Code.

Add a new file called 'code.fsx'.

Solving the Problem

This is where we left the code from the first chapter in this book:

```
type RegisteredCustomer = {
    Id : string
}

type UnregisteredCustomer = {
    Id : string
}

type Customer =
    | EligibleRegistered of RegisteredCustomer
    | Registered of RegisteredCustomer
    | Guest of UnregisteredCustomer

// Customer -> decimal -> decimal
let calculateTotal customer spend =
    let discount =
        match customer with
        | EligibleRegistered _ when spend >= 100.0M -> spend * 0.1M
        | _ -> 0.0M
    spend - discount
```

```
let john = EligibleRegistered { Id = "John" }
let mary = EligibleRegistered { Id = "Mary" }
let richard = Registered { Id = "Richard" }
let sarah = Guest { Id = "Sarah" }

let assertJohn = calculateTotal john 100.0M = 90.0M
let assertMary = calculateTotal mary 99.0M = 99.0M
let assertRichard = calculateTotal richard 100.0M = 100.0M
let assertSarah = calculateTotal sarah 100.0M = 100.0M
```

Copy the code into code.fsx.

It's nice but we still have some primitives where we should have domain concepts (Spend and Total). I would like the signature of the calculateTotal function to be (Customer -> Spend -> Total). The easiest way to achieve this is to use **Type Abbreviations**. Type abbreviations are simple aliases:

```
let TrueOrFalse = boolean
```

Anywhere we use a boolean, we could now use TrueOrFalse instead.

Add the following code below the Customer discriminated union:

```
type Spend = decimal
type Total = decimal
```

We have two approaches we can use with our new type abbreviations and get the type signature we need: Explicitly creating and implementing a function type which we did in Chapter 6 or using explicit types for the input/output parameter. Function type approach first:

```
type CalculateTotal = Customer -> Spend -> Total

//Customer -> Spend -> Total
let calculateTotal : CalculateTotal =
    fun customer spend ->
        let discount =
            match customer with
            | EligibleRegistered _ when spend >= 100.0M -> spend * 0.1M
            | _ -> 0.0M
        spend - discount
```

Note the change in the way that input parameters are used with the function type.

The same function using explicit parameters:

```
// Customer -> Spend -> Total
let calculateTotal (customer:Customer) (spend:Spend) : Total =
    let discount =
        match customer with
        | EligibleRegistered _ when spend >= 100.0M -> spend * 0.1M
        | _ -> 0.0M
    spend - discount
```

Either approach gives us the signature we want. I have no preference for the style as both are useful to know, so we will use the explicit parameters for the rest of this post.

There is a potential problem with type abbreviations: As long as the underlying type matches, I can use anything for the input, not just Spend.

There is nothing stopping you supplying either an invalid value or the wrong value to a parameter as shown in this example code:

```
type Latitude = decimal
type Longitude = decimal

type GpsCoordinate = Latitude * Longitude

// Latitude -90° to 90°
// Longitude -180° to 180°
let badGps : GpsCoordinate = (1000M, -345M)

let latitude = 46M
let longitude = 15M

// Swap latitude and longitude
let badGps2 : GpsCoordinate = (longitude, latitude)
```

We might be able to write tests to prevent this from happening but that is additional work. We want the type system to help us to write safer code. Thankfully there is a way that we can prevent these scenarios in F#: **The Single Case Discriminated Union**.

Let's define one for Spend by replacing the existing code with:

```
// type {typeName} = {typeConstructor} of {underlyingType}
type Spend = Spend of decimal
```

You will notice that the calculateTotal function now has errors. We can fix that by deconstructing the Spend parameter value in the function using pattern matching:

```
let calculateTotal (customer:Customer) (spend:Spend) : Total =
    let (Spend value) = spend
    let discount =
        match customer with
        | EligibleCustomer _ when value >= 100.0M -> value * 0.1M
        | _ -> 0.0M
    value - discount
```

If the type had been defined as ‘type Spend = xSpend of decimal’, the deconstructor would have been (xSpend value).

We need to change the asserts to use the new type constructor:

```
let assertJohn = calculateTotal john (Spend 100.0M) = 90.0M
let assertMary = calculateTotal mary (Spend 99.0M) = 99.0M
let assertRichard = calculateTotal richard (Spend 100.0M) = 100.0M
let assertSarah = calculateTotal sarah (Spend 100.0M) = 100.0M
```

The calculateTotal function is now safer but less readable. Thankfully, there is a simple fix for this, we can deconstruct the value in the function parameter directly:

```
// Customer -> decimal -> decimal
let calculateTotal customer (Spend spend) =
    let discount =
        match customer with
        | EligibleRegistered _ when spend >= 100.0M -> spend * 0.1M
        | _ -> 0.0M
    spend - discount
```

If you replace all of your primitives and type abbreviations with single case discriminated unions, you cannot supply the wrong parameter as the compiler will stop you. Of course it doesn’t stop the determined from abusing our code but it is another hurdle they must overcome.

The next improvement is to restrict the range of values that the Spend type can accept since very few domain values will be unbounded. We will restrict Spend to between 0.0M and 1000.0M. To support this, we are going to add a ValidationError type and prevent the direct use of the Spend constructor:

```
type ValidationError =
    | InputOutOfRange of string

type Spend = private Spend of decimal
    with
        member this.Value = this |> fun (Spend value) -> value
        static member Create input =
            if input >= 0.0M && input <= 1000.0M then
                Ok (Spend input)
            else
                Error (InputOutOfRange "You can only spend between 0 and 1000")
```

The use of the private accessor prevents code outside this module from directly accessing the type constructor. This means that to create an instance of Spend, we need to use the Spend.Create factory function defined on the type.

To extract the value, we use the Value property defined on the instance.

We need to make some changes to get the code to compile. Firstly we change the calculateTotal function:

```
let calculateTotal (customer:Customer) (spend:Spend) : Total =
    let discount =
        match customer with
        | EligibleRegistered _ when spend.Value >= 100.0M -> spend.Value * 0.1M
        | _ -> 0.0M
    spend.Value - discount
```

We also need to fix the asserts. To do this, we are going to add a new helper function to simplify the code:

```
let doCalculateTotal name amount =
    match Spend.Create amount with
    | Ok spend -> calculateTotal name spend
    | Error ex -> failwith (sprintf "%A" ex)

let assertJohn = doCalculateTotal john 100.0M = 90.0M
let assertMary = doCalculateTotal mary 99.0M = 99.0M
let assertRichard = doCalculateTotal richard 100.0M = 100.0M
let assertSarah = doCalculateTotal sarah 100.0M = 100.0M
```

We have done some extra work to the Spend type but by doing so we have ensured that an instance of it can never be invalid.

As a piece of homework, think about how you would use the features we have covered in this chapter on the code from the last chapter.

The final change that we can make is to move the discount rate from the calculateTotal function to be with the Customer type definition. The primary reason for doing this would be if we needed to use the discount in another function:

```
type Customer =
    | EligibleRegistered of RegisteredCustomer
    | Registered of RegisteredCustomer
    | Guest of UnregisteredCustomer
    with
        member this.Discount =
            match this with
            | EligibleRegistered _ -> 0.1M
            | _ -> 0.0M
```

This also allows us to simplify the calculateTotal function:

```
let calculateTotal (customer:Customer) (spend:Spend) : Total =
    let discount = if spend.Value >= 100.0M then spend.Value * customer.Discount els\
e 0.0M
    spend.Value - discount
```

Whilst this looks nice, it has broken the link between the customer type and the spend. Remember, the rule is a 10% discount if an eligible customer spends 100.0 or more. Let's have another go:

```
type Customer =
    | EligibleRegistered of RegisteredCustomer
    | Registered of RegisteredCustomer
    | Guest of UnregisteredCustomer
    with
        member this.CalculateDiscountPercentage(spend:Spend) =
            match this with
            | EligibleRegistered _ ->
                if spend.Value >= 100.0M then 0.1M else 0.0M
            | _ -> 0.0M
```

WARNING You will probably need to move the Spend type to before the Customer type declaration in the file.

We now need to modify our calculateTotal function to use our new function:

```
let calculateTotal (customer:Customer) (spend:Spend) : Total =
    let discount = spend.Value * customer.CalculateDiscountPercentage spend
    spend.Value - discount
```

To run this code, you will need to load all of it back into FSI using `ALT+ENTER` as we have changed quite a lot of code. Your tests should still pass.

A final change would be to simplify the calculateTotal function:

```
let calculateTotal (customer:Customer) (spend:Spend) : Total =
    spend.Value * (1.0M - customer.CalculateDiscountPercentage spend)
```

We have covered quite a few options in this chapter and all are valid. Which direction you go in is up to you.

Final Code

```
type RegisteredCustomer = {
    Id : string
}

type UnregisteredCustomer = {
    Id : string
}

type ValidationError =
    | InputOutOfRange of string

type Spend = private Spend of decimal
    with
        member this.Value = this |> fun (Spend value) -> value
        static member Create input =
            if input >= 0.0M && input <= 1000.0M then
                Ok (Spend input)
            else
                Error (InputOutOfRange "You can only spend between 0 and 1000")

type Total = decimal

type Customer =
    | EligibleRegistered of RegisteredCustomer
    | Registered of RegisteredCustomer
    | Guest of UnregisteredCustomer
    with
        member this.CalculateDiscountPercentage(spend:Spend) =
            match this with
            | EligibleRegistered _ ->
                if spend.Value >= 100.0M then 0.1M else 0.0M
            | _ -> 0.0M

let calculateTotal (customer:Customer) (spend:Spend) : Total =
    spend.Value * (1.0M - customer.CalculateDiscountPercentage spend)

let john = EligibleRegistered { Id = "John" }
let mary = EligibleRegistered { Id = "Mary" }
let richard = Registered { Id = "Richard" }
let sarah = Guest { Id = "Sarah" }

let doCalculateTotal name amount =
    match Spend.create amount with
    | Ok spend -> calculateTotal name spend
    | Error ex -> failwith (sprintf "%A" ex)

let assertJohn = doCalculateTotal john 100.0M = 90.0M
let assertMary = doCalculateTotal mary 99.0M = 99.0M
let assertRichard = doCalculateTotal richard 100.0M = 100.0M
let assertSarah = doCalculateTotal sarah 100.0M = 100.0M
```

Record Type

You can also use a record type to serve the same purpose as the single case discriminated union:

```
type Spend = private { Spend : decimal }
    with
        member this.Value = this.Spend
        static member Create input =
            if input >= 0.0M && input <= 1000.0M then
                Ok { Spend = input }
            else
                Error (InputOutOfRange "You can only spend between 0 and 1000")
```

Summary

In this post we have learnt about single case discriminated unions. They allows us to restrict the data that a parameter can accept compared to raw primitives. We have also seen that we can extend them by adding helper functions and properties.

In the next chapter, we will look at object programming in F#.

10 - Object Programming

In this post we are going to see how we can utilise some of the object programming features that F# offers. F# is a functional-first language but sometimes it's beneficial to use objects, particularly when interacting with code written in other, less functional .NET languages or when you want to encapsulate some internal data structures and/or mutable state.

F# can do pretty much anything that C#/VB.Net can do with objects. We are going to concentrate on the core object programming features; class types, interfaces, encapsulation, and equality.

Setting Up

Create a new folder for the chapter code and open it in VS Code.

Add three new files, FizzBuzz.fsx, RecentlyUsedList.fsx and Coordinate.fsx.

Class Types

We will start in FizzBuzz.fsx where we will be implementing FizzBuzz using object programming.

We are going to create our first class type and use the FizzBuzz function we created in Chapter 7:

```
type FizzBuzz() =
    member _.Calculate(value) =
        [(3, "Fizz");(5, "Buzz")]
        |> List.map (fun (v, s) -> if value % v = 0 then s else "")
        |> List.reduce (+)
        |> fun s -> if s <> "" then s else string value
```

Points of interest:

- The brackets () after the type name are required. They can contain arguments as we will see later in the chapter.
- The member keyword defines the accessible members of the type.
- The _ is just a placeholder - it can be anything. It is convention to use one of _, __ or *this*.

Now that we have created our class type, we need to instantiate it to use it.

```
let doFizzBuzz =
    let fizzBuzz = FizzBuzz()
    [1..15]
    |> List.map (fun n -> fizzBuzz.Calculate n)
```

We don't use *new* when creating an instance of a class type in F# except when it implements IDisposible<'T> and then we would use *use* instead of *let* to create a scope block.

The code can be simplified to:

```
let doFizzBuzz =
    let fizzBuzz = FizzBuzz()
    [1..15]
    |> List.map fizzBuzz.Calculate
```

At the moment, we can only use [(3, "Fizz");(5, "Buzz")] as the mapping but it is easy to pass the mapping in through the constructor:

```
type FizzBuzz(mapping) =
    member _.Calculate(value) =
        mapping
        |> List.map (fun (v, s) -> if n % v = 0 then s else "")
        |> List.reduce (+)
        |> fun s -> if s <> "" then s else string n
```

Notice that we don't need to assign the constructor argument to a binding to use it.

Now we need to pass the mapping in as the argument to the constructor in the doFizzBuzz function:

```
let doFizzBuzz =
    let fizzBuzz = FizzBuzz([(3, "Fizz");(5, "Buzz")])
    [1..15]
    |> List.map fizzBuzz.Calculate
```

We can move the function code from the member into the body of the class type as a new inner function:

```
type FizzBuzz(mapping) =
    let calculate n =
        mapping
        |> List.map (fun (v, s) -> if n % v = 0 then s else "")
        |> List.reduce (+)
        |> fun s -> if s <> "" then s else string n

    member _.Calculate(value) = calculate value
```

You cannot access the new calculate function from outside the class type. You don't have to do this but I find that it makes the code easier to read, especially as the number of class members increases.

Interfaces

Interfaces are very important in object programming as they define a contract that an implementation must handle. Let's create an interface to use in our fizzbuzz example:

```
// The 'I' prefix to the name is not required but is used by convention in .NET
type IFizzBuzz =
    abstract member Calculate : int -> string
```

Abstract Classes

To convert IFizzBuzz into an abstract class, decorate it with the [<AbstractClass>] attribute.

Now we need to implement the interface in our FizzBuzz class type:

```
type FizzBuzz(mapping) =
    let calculate n =
        mapping
        |> List.map (fun (v, s) -> if n % v = 0 then s else "")
        |> List.reduce (+)
        |> fun s -> if s <> "" then s else string n

    interface IFizzBuzz with
        member _.Calculate(value) = calculate value
```

Nice and easy but you will see that we have a problem; The compiler has highlighted the fizzBuzz.Calculate function call in our doFizzBuzz function.

```
let doFizzBuzz =
    let fizzBuzz = FizzBuzz([(3, "Fizz");(5, "Buzz")])
    [1..15]
    |> List.map (fun n -> fizzBuzz.Calculate(n)) // Problem
```

There is an error because F# does not support implicit casting, so we have to upcast the instance to the IFizzBuzz type ourselves:

```
let doFizzBuzz =
    let fizzBuzz = FizzBuzz([(3, "Fizz");(5, "Buzz")]) :> IFizzBuzz //Upcast
    [1..15]
    |> List.map (fun n -> fizzBuzz.Calculate(n)) // Fixed
```

An alternative would be to upcast as you use the interface function:

```
let doFizzBuzz =
    let fizzBuzz = FizzBuzz([(3, "Fizz");(5, "Buzz")])
    [1..15]
    |> List.map (fun n -> (fizzBuzz :> IFizzBuzz).Calculate(n))
```

If you have come from a language like C# where it supports implicit casting, it may seem odd to have to explicitly cast in F# but it gives you the safety of being certain about what your code is doing as you are not relying on compiler magic.

The code above is designed to show how to construct class types and use interfaces. If you find yourself constructing interfaces with one function, ask yourself if you really, really need the extra code and complexity or whether a simple function is enough.

There is another feature that interfaces offer that we haven't covered so far: Object Expressions.

Object Expressions

Create a new script file called expression.fsx and add the following import declaration:

```
open System
```

Now we are going to create a new interface:

```fsharp
type ILogger =
    abstract member Info : string -> unit
    abstract member Error : string -> unit
```

We would normally create a class type:

```fsharp
type Logger() =
    interface ILogger with
        member _.Info(msg) = printfn "Info: %s" msg
        member _.Error(msg) = printfn "Error: %s" msg
```

To use this, we need to create an instance and cast it to an ILogger.

Instead, we are going to create an object expression which simplifies the usage:

```fsharp
let logger = {
    new ILogger with
        member _.Info(msg) = printfn "Info: %s" msg
        member _.Error(msg) = printfn "Error: %s" msg
}
```

We have actually created an anonymous type, so we are restricted in what we can do with it. Let's create a class type and pass the ILogger in as a constructor argument:

```fsharp
type MyClass(logger:ILogger) =
    let mutable count = 0

    member _.DoSomething input =
        logger.Info $"Processing {input} at {DateTime.UtcNow.ToString()}"
        count <- count + 1
        ()

    member _.Count = count
```

Note that we can create members of the class type that are not associated with an interface.

We can now use the class type with the object expression logger:

```fsharp
let myClass = MyClass(logger)
[1..10] |> List.iter myClass.DoSomething
printfn "%i" myClass.Count
```

Run all of these blocks of code in FSI.

Create a function that takes in an ILogger as a parameter:

```
let doSomethingElse (logger:ILogger) input =
    logger.Info $"Processing {input} at {DateTime.UtcNow.ToString()}"
    ()
```

We could use our existing object expression logger but we can also create an object expression directly in the function call:

```
doSomethingElse {
    new ILogger with
        member _.Info(msg) = printfn "Info: %s" msg
        member _.Error(msg) = printfn "Error: %s" msg
    } "MyData"
```

This is a really useful feature if you have one-off services that you need to run.

Next we move on to a more complex example of using interfaces: A recently used list.

Encapsulation

We are going to create a recently used list as a class type. We will encapsulate a mutable collection within the class type and provide an interface for how we can interact with it. The recently used list is an ordered list with the most recent item first but it is also a set, so each value can only appear once in the list.

First we need to create an interface in RecentlyUsedList.fsx:

```
type IRecentlyUsedList =
    abstract member IsEmpty : bool
    abstract member Size : int
    abstract member Clear : unit -> unit
    abstract member Add : string -> unit
    abstract member Get : int -> string option
```

By looking at the signatures, we can see that IsEmpty and Size are read-only properties and Clear, Add and Get are functions. Now we can create our class type and implement the IRecentlyUsedList interface:

```
type RecentlyUsedList() =
    let items = ResizeArray<string>()

    let add item =
        items.Remove item |> ignore
        items.Add item

    let get index =
        if index >= 0 && index < items.Count
        then Some items.[items.Count - index - 1]
        else None

    interface IRecentlyUsedList with
        member _.IsEmpty = items.Count = 0
        member _.Size = items.Count
        member _.Clear() = items.Clear()
        member _.Add(item) = add item
        member _.Get(index) = get index
```

The ResizeArray<'T> is the F# synonym for the standard .NET mutable (List<'T>). Encapsulation ensures that you cannot access it directly, only via the public interface.

Let's test our code in FSI. Run each of the following lines separately:

```
let mrul = RecentlyUsedList() :> IRecentlyUsedList

mrul.Add "Test"

mrul.IsEmpty = false // Should return true

mrul.Add "Test2"
mrul.Add "Test3"
mrul.Add "Test"

mrul.Get(0) = Some "Test" // Should return true
```

Now let's add a maximum size (capacity) to our IRecentlyUsedList interface:

```
type IRecentlyUsedList =
    abstract member IsEmpty : bool
    abstract member Size : int
    abstract member Capacity : int
    abstract member Clear : unit -> unit
    abstract member Add : string -> unit
    abstract member Get : int -> string option
```

You will notice that the compiler is complaining that we haven't implemented all of the interface items, so let's fix it. Add the capacity as a constructor argument and implement the Add function to ensure the oldest item is removed if we are at capacity when adding a new item:

```
type RecentlyUsedList(capacity:int) =
    let items = ResizeArray<string>(capacity)

    let add item =
        items.Remove item |> ignore
        if items.Count = items.Capacity then items.RemoveAt 0
        items.Add item

    let get index =
        if index >= 0 && index < items.Count
        then Some items.[items.Count - index - 1]
        else None

    interface IRecentlyUsedList with
        member _.IsEmpty = items.Count = 0
        member _.Size = items.Count
        member _.Capacity = items.Capacity
        member _.Clear() = items.Clear()
        member _.Add(item) = add item
        member _.Get(index) = get index
```

Let's test our recently used list with capacity of 5 using FSI:

```
let mrul = RecentlyUsedList(5) :> IRecentlyUsedList

mrul.Capacity // Should be 5

mrul.Add "Test"
mrul.Size // Should be 1
mrul.Capacity // Should be 5

mrul.Add "Test2"
mrul.Add "Test3"
mrul.Add "Test4"
mrul.Add "Test"
mrul.Add "Test6"
mrul.Add "Test7"
mrul.Add "Test"

mrul.Size // Should be 5
mrul.Capacity // Should be 5
mrul.Get(0) = Some "Test" // Should return true
mrul.Get(4) = Some "Test3" // Should return true
```

Encapsulation inside class types works really nicely, even for mutable data.

Now we move on to the issue of equality.

Equality

Most of the types in F# support structural equality but class types do not: They rely on reference equality like most things in .NET.

Create a simple class type to store GPS coordinates in Coordinate.fsx:

```
type Coordinate(latitude: float, longitude: float) =
    member _.Latitude = latitude
    member _.Longitude = longitude
```

To test equality, we can write some simple checks we can run in FSI:

```
let c1 = Coordinate(25.0, 11.98)
let c2 = Coordinate(25.0, 11.98)
let c3 = c1
c1 = c2 // false
c1 = c3 // true - reference the same instance
```

To support something like structural equality, we need to override the GetHashCode and Equals functions, implement IEquatable<'T> and if we are going to use it with other .NET languages, we need to handle the '=' operator using op_Equality and set the Allow Null Literal attribute:

```
open System

[<AllowNullLiteral>]
type GpsCoordinate(latitude: float, longitude: float) =
    let equals (other: GpsCoordinate) =
        if isNull other then
            false
        else
            latitude = other.Latitude
            && longitude = other.Longitude

    member _.Latitude = latitude
    member _.Longitude = longitude

    override this.GetHashCode() =
        hash (this.Latitude, this.Longitude)

    override _.Equals(obj) =
        match obj with
        | :? GpsCoordinate as other -> equals other
        | _ -> false

    interface IEquatable<GpsCoordinate> with
        member _.Equals(other: GpsCoordinate) =
            equals other

    static member op_Equality(this: GpsCoordinate, other: GpsCoordinate) =
        this.Equals(other)
```

We have used two built-in functions: *hash* and *isNull*. We make use of pattern matching in the Equals function by using a match expression to see if the object passed in can be cast as a GpsCoordinate and if it can, it is checked for equality.

If we test this, we get the expected structural equality:

```
let c1 = GpsCoordinate(25.0, 11.98)
let c2 = GpsCoordinate(25.0, 11.98)
c1 = c2 // true
```

We have only scratched the surface of what is possible with F# object programming. If you want to find out more about this topic (plus many other useful things), I highly recommend that you read Stylish F#[22] by Kit Eason[23].

Summary

In this chapter, we have had an introduction to object programming in F#. In particular, we have looked at scoping/visibility, encapsulation, interfaces/casting, and equality. The more you interact with the rest of the .NET ecosystem, the more you will potentially need to use object programming.

In the next chapter we will discover how F# handles recursion.

[22]https://www.apress.com/us/book/9781484239995
[23]https://twitter.com/kitlovesfsharp

11 - Recursion

In this chapter, we are going to look at recursive functions, that is functions that call themselves in a loop. We will start with a naive implementation and then make it more efficient using an accumulator. As a special treat, I will show you a way of writing FizzBuzz and an elegant quicksort algorithm using this technique.

Setting Up

Create a new folder for the code in this chapter.

All of the code in this chapter can be run using FSI from a .fsx file that you create.

Solving The Problem

We are going to start with a naive implementation of the factorial function (!):

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

To create a recursive function, we use the *rec* keyword and we would create a function like this:

```
// int -> int
let rec fact n =
    match n with
    | 1 -> 1
    | n -> n * fact (n-1)
```

You'll notice that we have two cases: 1 and greater than 1. When n is 1, we return 1 and the recursion completes but if it is greater than 1, we return the number multiplied by the factorial of the number - 1 and continue the recursion. If we were to write out what happens, it would look like this:

```
fact 5 = 5 * fact 4
       = 5 * (4 * fact 3)
       = 5 * (4 * (3 * fact 2))
       = 5 * (4 * (3 * (2 * fact 1)))
       = 5 * (4 * (3 * (2 * 1)))
       = 5 * (4 * (3 * 2))
       = 5 * (4 * 6)
       = 5 * 24
       = 120
```

This is a problem because you can't perform any calculations until you've completed all of the iterations but you also need to store all of the parts as well. This means that the larger n gets, the more memory you need to perform the calculation and it can also lead to stack overflows. We can solve this problem with **Tail Call Optimisation**.

Tail Call Optimisation

There are a few possible approaches available but we are going to use an accumulator. The accumulator is passed around the recursive function on each iteration:

```
let fact n =
    let rec loop n acc =
        match n with
        | 1 -> acc
        | _ -> loop (n-1) (acc * n)
    loop n 1
```

We leave the public interface of the function intact but create a function enclosed in the outer one to do the recursion. We also need to add a line at the end of the function to start the recursion and return the result. You'll notice that we've added an additional parameter (acc) which holds the accumulated value. As we are multiplying, we need to initialise the accumulator to 1. If the accumulator used addition, we would set it to 0 initially.

Lets write out what happens when we run this function as we did the previous version in pseudocode:

```
fact 5 = loop 5 1
       = loop 4 5
       = loop 3 20
       = loop 2 60
       = loop 1 120
       = 120
```

This is much simpler than the previous example, requires very little memory and is extremely efficient.

We can also solve factorial using the List.reduce and List.fold functions from the List module. The difference is that fold has an initial value and reduce uses only the values in the list:

```
let fact n = [1..n] |> List.fold (fun acc n -> acc * n) 1
let fact n = [1..n] |> List.reduce (fun acc n -> acc * n)
```

The F# tooling in VS Code will inform you that the accumulator functions can be shortened to:

```
let fact n = [1..n] |> List.fold (*) 1
let fact n = [1..n] |> List.reduce (*)
```

Now that we've learnt about tail call optimisation using an accumulator, let's look at a harder example, the Fibonacci Sequence.

Expanding the Accumulator

The Fibonacci Sequence is a simple list of numbers where each value is the sum of the previous two:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

Let's start with the naive example to calculate the nth item in the sequence:

```
let rec fib (n:int64) =
    match n with
    | 0L -> 0L
    | 1L -> 1L
    | s -> fib (s-1L) + fib (s-2L)
```

If you want to see just how inefficient this is, try running fib 50L. It will take almost a minute on a fast machine! Let's have a go at writing a more efficient version that uses tail call optimisation:

```
let fibL (n:int64) =
    let rec loop n (a,b) =
        match n with
        | 0L -> a
        | 1L -> b
        | n -> loop (n-1L) (b, a+b)
    loop n (0L,1L)
```

Let's write out what happens (ignoring the type annotation):

```
fibL 5L = loop 5L (0L, 1L)
        = loop 4L (1L, 1L+0L)
        = loop 3L (1L, 1L+1L)
        = loop 2L (2L, 1L+2L)
        = loop 1L (3L, 2L+3L)
        = 3L
```

The 5th item in the sequence is indeed 3 as the list starts at index 0. Try running fibL 50L - It should return almost instantaneously.

Next, we'll now continue on our journey to find as many functional ways of solving FizzBuzz as possible. :)

FizzBuzz Using Recursion

We start with a list of rules that we are going to recurse over:

```
let fizzRules =
    [ (3, "Fizz"); (5, "Buzz") ]
```

Our fizzbuzz function using tail call optimisation and has an accumulator that will use string concatenation and an initial value of "".

```
let fizzBuzz initialRules n =
    let rec loop remainingRules acc =
        match remainingRules, acc with
        | [], "" -> string n
        | [], _ -> acc
        | head::tail, _ ->
            let value =
                head |> (fun (i, v) -> if n % i = 0 then v else "")
            loop tail (acc + value)
    loop initialRules ""
```

The pattern match is asking:

1. If there are no rules left and the accumulator is “”, return the original input as a string
2. If there are no rules left, return the accumulator.
3. If there are rules left, loop with the tail as the rules and add the result of the function to the accumulator.

Finally we can run the function and print out the results to the terminal:

```
[ 1 .. 105 ]
|> List.map (fizzBuzz fizzRules)
|> List.iter (printfn "%s")
```

This is quite a nice extensible approach to the FizzBuzz problem as adding 7-Bazz is as easy as adding another rule.

```
let fizzRules =
    [ (3, "Fizz"); (5, "Buzz"); (7, "Bazz") ]
```

Having produced a nice solution to the FizzBuzz problem using recursion, can we use the List.fold function to solve it? Of course we can!

```
let fizzBuzz n =
    [ (3, "Fizz"); (5, "Buzz") ]
    |> List.fold (fun acc (value,name) ->
        if n % value = 0 then acc + name else acc) ""
    |> fun s -> if s = "" then string n else s

[1..105]
|> List.iter (fizzBuzz >> printfn "%s")
```

We can modify the code to do all of the mapping in the fold function rather than passing the value on to another function:

```
let fizzBuzz n =
    [ (3, "Fizz"); (5, "Buzz") ]
    |> List.fold (fun acc (value,name) ->
        match (if n % value = 0 then name else "") with
        | "" -> acc
        | s -> if acc = string n then s else acc + s) (string n)
```

The List.fold function is a very useful one to know well.

Lets have a look at another favourite algorithm, quicksort.

Quicksort using recursion

Quicksort[24] is a nice algorithm to create in F# because of the availability of some very useful collection functions in the List module:

```
let rec qsort input =
    match input with
    | [] -> []
    | head::tail ->
        let smaller, larger = List.partition (fun n -> head >= n) tail
        List.concat [qsort smaller; [head]; qsort larger]

[5;9;5;2;7;9;1;1;3;5] |> qsort |> printfn "%A"
```

The List.partition function splits a list into two based on a predicate function, in this case items smaller than or equal to the head value and those larger than it. List.concat converts a deeply nested sequence of lists into a single list<int>.

[24]https://www.tutorialspoint.com/data_structures_algorithms/quick_sort_algorithm.htm

Other Uses Of Recursion

The examples we've used are necessarily simple to concentrate on tail call optimisation using an accumulator but what would we use recursion for in a real-world application? Recursion is great for handling hierarchical data like the file system and XML, converting between flat data and hierarchies and for event loops or loops where there is no defined finish.

Further Reading

As always, Scott Wlaschin's excellent site[25] has many posts on the topic .

Summary

In this post we have had a look at recursion in F#. In particular, we have looked at how to use accumulators with tail call optimisation to make recursion more efficient.

In the next chapter, we will investigate Computation Expressions.

[25]https://fsharpforfunandprofit.com/posts/recursive-types-and-folds-3b/#series-toc

12 - Computation Expressions

In this chapter we are going to look at computation expressions including a first look at asynchronous code in F#. Computation expressions are syntactic sugar for simplifying code around effects like Option, Result and Async.

In this post we will start to learn how to create our own simple computation expression, how to use it, and look at a more complex example where we combine two effects together: Async and Result.

Setting Up

Create a new folder in VS Code, open a new Terminal to create a new console app using:

```
dotnet new console -lang F#
```

Using the Explorer view, create a folder called resources in the code folder for this chapter and then create a new file called customers.csv. Copy the following data into the new file:

```
CustomerId|Email|Eligible|Registered|DateRegistered|Discount
John|john@test.com|1|1|2015-01-23|0.1
Mary|mary@test.com|1|1|2018-12-12|0.1
Richard|richard@nottest.com|0|1|2016-03-23|0.0
Sarah||0|0||
```

Introducing the Problem

Add a new file above Program.fs called OptionDemo.fs. In the new file, copy the following code:

```
namespace ComputationExpression

module OptionDemo =

    let multiply x y = // int -> int -> int
        x * y

    let divide x y = // int -> int -> int option
        if y = 0 then None
        else Some (x / y)

    let calculate x y =
        divide x y
        |> function
            | Some v -> multiply v x |> Some
            | None -> None
        |> function
            | Some t -> divide t y
            | None -> None
```

We have two simple functions, multiply and divide, and we use them to compose a bigger function called calculate. To use a match expression directly in a pipeline, we can replace the *match … with* with *function*. This function looks ugly with all of the Option pattern matching. Thankfully, we know that we can use the Option module to simplify it:

```
let calculate x y =
    divide x y
    |> Option.map (fun s -> multiply s x)
    |> Option.bind (fun x -> divide x y)
```

Much nicer but what would it look like using a computation expression?

First we will create a basic one. Copy the following code between the namespace and the module declaration in OptionDemo.fs:

```
[<AutoOpen>]
module Option =

    type OptionBuilder() =
        // Supports let!
        member _.Bind(x, f) = Option.bind f x
        // Supports return
        member _.Return(x) = Some x
        // Supports return!
        member _.ReturnFrom(x) = x

    // Computation Expression for Option
    // Usage will be option {...}
    let option = OptionBuilder()
```

The [<AutoOpen>] attribute means that when the namespace is referenced, we automatically get access to the types and functions in the module.

This is a very simple computation expression but is enough for our needs. It's not important to know how this works but you can see that we define a class type with some required member functions and then create an instance of that type for use in our code.

Replace the calculate function with the following that uses the option computation expression:

```
let calculate x y =
    option {
        let! first = divide x y
        let second = multiply first x
        let! third = divide second y
        return third
    }
```

The nice thing about this function is that we don't need to know about Option.bind or Option.map; It's nearly all hidden away from us, so we can concentrate on the functionality.

If you hover your mouse over the first value, it tells you that you that it is an int because the bang (!) has unwrapped the option for us. Delete the bang and hover over the first value again. You'll see that it is now an option<int>, so it hasn't been unwrapped. You will also get an error from the compiler. Put the bang back.

The bindings with the *let!* get processed by the Bind function in the computation expression. Let bindings that would have used Option.map are automatically handled by the underlying computation expression code. The return matches the Return function in the computation expression. If we want to use the ReturnFrom function, we do the following:

```
let calculate x y =
    option {
        let! first = divide x y
        let second = multiply first x
        return! divide second y
    }
```

Notice the bang (!) on the return.

To test the code, replace the code in Program.fs with the following:

```
open ComputationExpression.OptionDemo

[<EntryPoint>]
let main argv =
    calculate 8 0 |> printfn "calculate 8 0 = %A"
    calculate 8 2 |> printfn "calculate 8 2 = %A"
    0
```

Leave the 0 at the end to signify the app has completed successfully.

Now type dotnet run in the Terminal. You should get None and Some 16 as the output in the terminal window.

The Result Computation Expression

Create a new file ResultDemo.fs above Program.fs.

Rather than create our own computation expression for Result, we will use an existing one from the FsToolkit.ErrorHandling NuGet package.

Use the Terminal to add a NuGet package that contains the result computation expression that we want to use.

```
dotnet add package FsToolkit.ErrorHandling
```

Copy the following code into ResultDemo.fs:

```
namespace ComputationExpression

module ResultDemo =

    open FsToolkit.ErrorHandling

    type Customer = {
        Id : int
        IsVip : bool
        Credit : decimal
    }

    let getPurchases customer = // Customer -> Result<(Customer * decimal),exn>
        try
            // Imagine this function is fetching data from a Database
            let purchases =
                if customer.Id % 2 = 0 then (customer, 120M) else (customer, 80M)
            Ok purchases
        with
        | ex -> Error ex

    let tryPromoteToVip purchases = // Customer * decimal -> Customer
        let customer, amount = purchases
        if amount > 100M then { customer with IsVip = true }
        else customer

    let increaseCreditIfVip customer = // Customer -> Result<Customer,exn>
        try
            // Imagine this function could cause an exception
            let increase = if customer.IsVip then 100M else 50M
            Ok { customer with Credit = customer.Credit + increase }
        with
        | ex -> Error ex

    let upgradeCustomer customer =
        customer
        |> getPurchases
        |> Result.map tryPromoteToVip
        |> Result.bind increaseCreditIfVip
```

Now let's see what turning the upgradeCustomer function into a computation expression looks like by replacing it with the following:

```
let upgradeCustomer customer =
    result {
        let! purchases = getPurchases customer
        let promoted = tryPromoteToVip purchases
        return! increaseCreditIfVip promoted
    }
```

Notice that the bang (!) is only applied to those functions that apply the effect, in this case Result.

I find this style easier to read but some people do prefer the previous version.

Introduction to Async

Create a new file AsyncDemo.fs above Program.fs and create the following code:

```
namespace ComputationExpression

module AsyncDemo =

    open System.IO

    type FileResult = {
        Name: string
        Length: int
    }

    let getFileInformation path =
        async {
            let! bytes = File.ReadAllBytesAsync(path) |> Async.AwaitTask
            let fileName = Path.GetFileName(path)
            return { Name = fileName; Length = bytes.Length }
        }
```

The Async.AwaitTask function call is required because File.ReadAllBytesAsync returns a Task because it is a .NET function rather than an F# one. It is very similar in style to Async/Await in C# but Async in F# is lazily evaluated and Task is not.

Let's test our new code. Add the following import declaration in Program.fs:

```
open ComputationExpression.AsyncDemo
```

Replace the code in the main function with the following code:

```
Path.Combine(__SOURCE_DIRECTORY__, "resources", "customers.csv")
|> getFileInformation
|> Async.RunSynchronously
|> printfn "%A"
0
```

We have to use Async.RunSynchronously to force the async code to run. You shouldn't do this in production code.

Run the code by typing the following in the terminal window:

```
dotnet run
```

Computation Expressions in Action

So far we have seen how to use a single effect but what happens if we have to use two (or more) like Async and Result? Thankfully, such a thing is possible and quite commonplace in this style of programming. We are going to use the asyncResult computation expression from the FsToolkit.ErrorHandling NuGet package we installed earlier.

The original code for this example comes from FsToolkit.ErrorHandling[26].

Create a new file called AsyncResultDemo.fs above Program.fs and add the following code:

```
namespace ComputationExpression

module AsyncResultDemo =

    open System
    open FsToolkit.ErrorHandling

    type AuthError =
        | UserBannedOrSuspended

    type TokenError =
        | BadThingHappened of string

    type LoginError =
        | InvalidUser
        | InvalidPwd
        | Unauthorized of AuthError
        | TokenErr of TokenError
```

[26]https://demystifyfp.gitbook.io/fstoolkit-errorhandling/asyncresult/ce

```
    type AuthToken = AuthToken of Guid

    type UserStatus =
        | Active
        | Suspended
        | Banned

    type User = {
        Name : string
        Password : string
        Status : UserStatus
    }
```

AuthToken is a single case discriminated union. It could have been written like TokenError but is generally written as seen.

Add some constants below the type definitions, marked with the [<Literal>] attribute:

```
[<Literal>]
let ValidPassword = "password"
[<Literal>]
let ValidUser = "isvalid"
[<Literal>]
let SuspendedUser = "issuspended"
[<Literal>]
let BannedUser = "isbanned"
[<Literal>]
let BadLuckUser = "hasbadluck"
[<Literal>]
let AuthErrorMessage = "Earth's core stopped spinning"
```

Now we add the core functions, some of which are Async and some return Results. Don't worry about the how the functions are implementated internally, concentrate only on the inputs and outputs:

```
let tryGetUser (username:string) : Async<User option> =
    async {
        let user = { Name = username; Password = ValidPassword; Status = Active }
        return
            match username with
            | ValidUser -> Some user
            | SuspendedUser -> Some { user with Status = Suspended }
            | BannedUser -> Some { user with Status = Banned }
            | BadLuckUser -> Some user
            | _ -> None
    }

let isPwdValid (password:string) (user:User) : bool =
    password = user.Password

let authorize (user:User) : Async<Result<unit, AuthError>> =
    async {
        return
            match user.Status with
            | Active -> Ok ()
            | _ -> UserBannedOrSuspended |> Error
    }

let createAuthToken (user:User) : Result<AuthToken, TokenError> =
    try
        if user.Name = BadLuckUser then failwith AuthErrorMessage
        else Guid.NewGuid() |> AuthToken |> Ok
    with
    | ex -> ex.Message |> BadThingHappened |> Error
```

The final part is to add the main login function that uses the previous functions and the asyncResult computation expression. The function does four things - tries to get the user from the datastore, checks the password is valid, checks the authorization and creates a token to return:

```
let login (username: string) (password: string) : Async<Result<AuthToken, LoginError\
>> =
    asyncResult {
        let! user = username |> tryGetUser |> AsyncResult.requireSome InvalidUser
        do! user |> isPwdValid password |> Result.requireTrue InvalidPwd
        do! user |> authorize |> AsyncResult.mapError Unauthorized
        return! user |> createAuthToken |> Result.mapError TokenErr
    }
```

The return type from the function is Async<Result<AuthToken, LoginError>>, so asyncResult is Async wrapping Result. This is very common in Line of Business (LOB) applications written in F#.

Notice the use of functions from the referenced library which make the code nice and succinct. The 'do! = ...' supports functions that return unit and is the same as writing 'let! _ = ...' where we would ignore the returned value. The Result.mapError functions are used to convert the specific errors from the authorize and createAuthToken functions to the LoginError type used by the login function.

To test the code, copy the following under the existing code in AsyncResultDemo.fs or create a new file called AsyncResultDemoTests.fs below AsyncResultDemo.fs and above Program.fs.

```
module AsyncResultDemoTests =

    open AsyncResultDemo

    [<Literal>]
    let BadPassword = "notpassword"
    [<Literal>]
    let NotValidUser = "notvalid"

    let isOk (input:Result<_,_>) : bool =
        match input with
        | Ok _ -> true
        | _ -> false

    let matchError (error:LoginError) (input:Result<_,LoginError>) =
        match input with
        | Error ex -> ex = error
        | _ -> false

    let runWithValidPassword (username:string) =
        login username ValidPassword |> Async.RunSynchronously

    let success =
        let result = runWithValidPassword ValidUser
        result |> isOk

    let badPassword =
        let result = login ValidUser BadPassword |> Async.RunSynchronously
        result |> matchError InvalidPwd

    let invalidUser =
        let result = runWithValidPassword NotValidUser
        result |> matchError InvalidUser

    let isSuspended =
        let result = runWithValidPassword SuspendedUser
        result |> matchError (UserBannedOrSuspended |> Unauthorized)

    let isBanned =
        let result = runWithValidPassword BannedUser
        result |> matchError (UserBannedOrSuspended |> Unauthorized)

    let hasBadLuck =
        let result = runWithValidPassword BadLuckUser
        result |> matchError (AuthErrorMessage |> BadThingHappened |> TokenErr)
```

In the Program.fs main function copy the following:

```
printfn "Success: %b" success
printfn "BadPassword: %b" badPassword
printfn "InvalidUser: %b" invalidUser
printfn "IsSuspended: %b" isSuspended
printfn "IsBanned: %b" isBanned
printfn "HasBadLuck: %b" hasBadLuck
```

In the terminal, type dotnet run and press enter. You should see successful asserts.

The code for this section is available here[27].

Debugging Code

The observant amongst you will have noticed that Ionide supports debugging. It's something that we don't do very often as pure functions and F# Interactive (FSI) are more convenient. You can add a breakpoint like you do in other IDEs by clicking in the border to the left of the code.

[27]https://gist.github.com/ianrussellsoftwarepark/2d11367c69d5f14231d439580034742d

If you put a breakpoint in the createAuthToken function on the if expression and debug the code by pressing the green button, you will see that you only hit the breakpoint twice, on the success and hasbadluck cases. Once the running code is on the Error track, it won't run any of the remaining code on the Ok track.

Further Reading

Computation Expressions are very useful tools to have at your disposal and are well worth investigating further.

It is vital that you learn more about how F# handles asynchronous code with async and how it interacts with the Task type from .Net Core. You can find out more at the following link:

https://docs.microsoft.com/en-us/dotnet/fsharp/tutorials/asynchronous-and-concurrent-programming/async

Summary

In this chapter we have had a look at computation expressions in F#. They can be a little confusing to begin with but make for extremely readable code.

In the next chapter, we are going to start to put some of the skills we have developed throughout this book into practical use by looking at how to create APIs and websites with Giraffe.

13 - Introduction to Web Programming with Giraffe

In this chapter we will start investigating web programming in F#.

We are going to use Giraffe[28]. Giraffe is "an F# micro web framework for building rich web applications" and was created by Dustin Moris Gorski[29]. It is a thin functional wrapper around ASP.NET Core. Both APIs and web pages are supported by Giraffe and we will cover both in the last three chapters of this book.

Giraffe comes with excellent documentation[30]. I highly recommend that you spend some time looking through it to see what feature Giraffe offers.

If you want something more opinionated or want to use F# everywhere including to create JavaScript, have a look at Saturn[31] and the Safe Stack[32].

Getting Started

Create a new folder called GiraffeExample and open it in VS Code.

Using the Terminal in VS Code, type in the following command to create an empty ASP.NET Core web app project:

```
dotnet new web -lang F#
```

Add the following NuGet packages from the terminal:

```
dotnet add package Giraffe
dotnet add package Giraffe.ViewEngine
```

Open Startup.fs. If you've done any ASP.NET Core development before, it will look very familiar, even though it is in F#:

[28]https://github.com/giraffe-fsharp/Giraffe
[29]https://twitter.com/dustinmoris
[30]https://github.com/giraffe-fsharp/Giraffe/blob/master/DOCUMENTATION.md
[31]https://saturnframework.org/
[32]https://safe-stack.github.io/

```
type Startup() =

    // This method gets called by the runtime. Use this method to add services to th\
e container.
    // For more information on how to configure your application, visit https://go.m\
icrosoft.com/fwlink/?LinkID=398940
    member _.ConfigureServices(services: IServiceCollection) =
        ()

    // This method gets called by the runtime. Use this method to configure the HTTP\
 request pipeline.
    member _.Configure(app: IApplicationBuilder, env: IWebHostEnvironment) =
        if env.IsDevelopment() then
            app.UseDeveloperExceptionPage() |> ignore

        app.UseRouting()
           .UseEndpoints(fun endpoints ->
                endpoints.MapGet("/", fun context ->
                    context.Response.WriteAsync("Hello World!")) |> ignore
            ) |> ignore
```

We need to make some changes to plug in Giraffe. Add the following import declarations below the ones already there:

```
open Giraffe
open Giraffe.ViewEngine
open FSharp.Control.Tasks
```

Replace unit () in the ConfigureServices function with:

```
services.AddGiraffe() |> ignore
```

Replace the app.UseRouting code in the Configure function with:

```
app.UseGiraffe (route "/" >=> text "Hello World!")
```

Running the Sample Code

In the Terminal, type the following to run the project:

```
dotnet run
```

Go to your browser and type in the following Url:

https://localhost:5001

You should see “Hello World!” in your browser window.

To shut the running website down, press CTRL+C in the Terminal window where the code is running. You should be back to the command prompt.

Before we add new functionality, we need to understand what is going on in (route “/” >=> text “Hello World!”).

HttpHandlers and Combinators

Let’s take a look at the Giraffe sourcecode for *route* and *text*. They are both functions that return HttpHandler. The route function takes the path as input and the text function takes the message as input:

```
let route (path : string) : HttpHandler =
    fun (next : HttpFunc) (ctx : HttpContext) ->
        if (SubRouting.getNextPartOfPath ctx).Equals path
        then next ctx
        else skipPipeline

let text (str : string) : HttpHandler =
    let bytes = Encoding.UTF8.GetBytes str
    fun (_ : HttpFunc) (ctx : HttpContext) ->
        ctx.SetContentType "text/plain; charset=utf-8"
        ctx.WriteBytesAsync bytes
```

The route HttpHandler checks the input path against the one from the route parameter and if they match, it passes control to the next handler in that pipeline otherwise it exits this pipeline and hands control back to routing to try the next route.

The text HttpHandler simply writes to the response and sets the response content type.

Pretty much everything you are going to see in Giraffe for routing, middleware etc is a HttpHandler.

The >=> operator represents a combinator. It looks and sounds scary but, like most things in F#, it looks far worse than it is! It is a custom operator to use instead of the compose function. It is defined like this:

```
let (>=>) = compose
```

I'm not going to show the code for the compose function but it takes two HttpHandlers as input and returns a HttpHandler as output so that the result can also be composed with other HttpHandlers in a pipeline.

The '(>⇒)' is a custom operator that is used as a type abbreviation for the compose function.

We can use the compose function in one of three ways:

```
// Using the compose function directly
let webApp = compose (route "/") (text "Hello World!")

// Using the custom operator directly
let webApp = (>=>) (route "/") (text "Hello World!")

// Using the infix version of the custom operator
let webApp = route "/" >=> text "Hello World!"
```

I don't normally like custom operators because they can make it harder to quickly grasp what the code is doing but in this case, the infix version of the custom operator makes the both code easier to read and to use.

Next Steps

It's a bit limiting only having one route in your app which we do at the moment, so we'll start to rectify that by moving the current route out to a new module. Create a new module called Handlers between the import declarations and the Startup class type definition and move the webApp binding into it:

```
module Handlers =

    let webApp = route "/" >=> text "Hello World!"
```

Replace the route used in app.UseGiraffe with:

```
app.UseGiraffe Handlers.webApp
```

Run the app again using dotnet run to ensure it is still working as expected.

There are many other HttpHandlers built into Giraffe along with another combinator called *choose* that allows us to define a list of potential routes:

```
    let webApp =
        choose [
            route "/" >=> text "Hello World!"
            setStatusCode 404 >=> text "Not Found"
        ]
```

The routing will choose the first route that the request can complete.

We should also restrict the HTTP verb to GET requests only. HTTP verbs in Giraffe are HttpHandlers, so they can be composed in the pipeline. Anything other than 'GET /' is going to result in a 404 error:

```
    let webApp =
        choose [
            GET >=> route "/" >=> text "Hello World!"
            RequestErrors.NOT_FOUND "Not Found"
        ]
```

The setStatusCode 404 can be replaced by a built-in helper to simplify it.

Creating an API

We are going to add a new route '/api' and respond with some JSON using another built-in Giraffe HttpHandler, *json*:

```
let webApp =
    choose [
        GET >=> route "/" >=> text "Hello World!"
        GET >=> route "/api" >=> json {| Response = "Hello world!!" |}
        RequestErrors.NOT_FOUND "Not Found"
    ]
```

The payload for the JSON response on the '/api' route is an anonymous record, designated by the {|..|} structure. You define the structure and data of the anonymous record in place rather than defining a type and creating an instance.

Now run the code and try the following Url in the browser. You should see a JSON response:

https://localhost:5001/api

Creating a Custom HttpHandler

There are lots of built-in HttpHandlers but it is simple to create your own because they are just functions. Let's add a new route for the API that takes a string value in the request querystring that is handled by a custom handler that we haven't written yet:

```
let webApp =
    choose [
        GET >=> route "/" >=> text "Hello World!"
        GET >=> route "/api" >=> json {| Response = "Hello world!!" |}
        GET >=> routef "/api/%s" sayHelloNameHandler
        RequestErrors.NOT_FOUND "Not Found"
    ]
```

Note that the querystring item is automatically bound to the handler function if it has a matching input parameter of the same type in the handler, so the sayHelloNameHandler needs to take a string input parameter. Create the custom handler in the Handlers module above the webApp binding:

```
let sayHelloNameHandler (name:string) : HttpHandler =
    fun (next:HttpFunc) (ctx:HttpContext) ->
        task {
            let msg = $"Hello {name}, how are you?"
            return! json {| Response = msg |} next ctx
        }
```

This function could be rewritten as the following but it is convention to use the style above:

```
let sayHelloNameHandler (name:string) (next:HttpFunc) (ctx:HttpContext) : HttpHandle\
r =
    task {
        let msg = $"Hello {name}, how are you?"
        return! json {| Response = msg |} next ctx
    }
```

The task {…} is a computation Expression of type System.Threading.Tasks.Task<'T> and is the equivalent to async/await in C#. F# has another asynchronous feature called async but we won't use that as Giraffe has chosen to work directly with Task instead.

You will see a lot more of the task computation expression in the next chapter as we expand the API side of our website.

Now run the code and try the following Url in the browser. If you replace the placeholder ‘’ with your name, you should see a JSON response asking how you are:

https://localhost:5001/api/\protect\char"007B\relaxyourname\protect\char"007D\relax

Not only is Giraffe excellent as an API, it is equally suited to server-side rendered web pages using the Giraffe View Engine.

Creating a View

The Giraffe View Engine is a DSL that generates HTML. This is what a simple page looks like:

```
let indexView =
    html [] [
        head [] [
            title [] [ str "Giraffe Example" ]
        ]
        body [] [
            h1 [] [ str "I |> F#" ]
            p [ _class "some-css-class"; _id "someId" ] [
                str "Hello World from the Giraffe View Engine"
            ]
        ]
    ]
```

Each element of html has the following structure:

```
// element [attributes] [sub-elements/data]
html [] []
```

Each element has two supporting lists: The first is for attributes and the second for sub-elements or data. You may wonder why we need a DSL to generate HTML but it makes sense as it helps prevent badly formed structures. All of the features you need like master pages, partial views and model binding are included. We will have a deeper look at views in the final chapter of this book.

Add the indexView value code above the webApp handler and replace the root route with the following:

```
GET >=> route "/" >=> htmlView indexView
```

Run the web app and you will see the text from the indexView in your browser.

Tidying up Routing

One last thing to do and that is to simplify the routing as we have quite a lot of duplicate code:

```
let webApp =
    choose [
        GET >=> route "/" >=> htmlView indexView
        GET >=> route "/api" >=> json {| Response = "Hello world!!" |}
        GET >=> routef "/api/%s" sayHelloNameHandler
        RequestErrors.NOT_FOUND "Not Found"
    ]
```

The first thing we do is extract the GET HttpHandler and add a choose combinator:

```
let webApp =
    choose [
        GET >=> choose [
            route "/" >=> htmlView indexView
            route "/api" >=> json {| Response = "Hello world!!" |}
            routef "/api/%s" sayHelloNameHandler
        ]
        RequestErrors.NOT_FOUND "Not Found"
    ]
```

We can extract routes to another HttpHandler like we do with the '/api' route here:

```
let apiRoutes : HttpHandler =
    choose [
        route "" >=> json {| Response = "Hello world!!" |}
        routef "/%s" sayHelloNameHandler
    ]

let webApp =
    choose [
        GET >=> choose [
            route "/" >=> htmlView indexView
            subRoute "/api" apiRoutes
        ]
        RequestErrors.NOT_FOUND "Not Found"
    ]
```

Note the change from 'route' to 'subRoute' if we extract the '/api' route to its own handler.

Run the app and try the routes out in your browser.

Reviewing the Code

The finished code for this chapter is:

```
namespace GiraffeExample

open System
open Microsoft.AspNetCore.Builder
open Microsoft.AspNetCore.Hosting
open Microsoft.AspNetCore.Http
open Microsoft.Extensions.DependencyInjection
open Microsoft.Extensions.Hosting
open Giraffe
open Giraffe.ViewEngine
open FSharp.Control.Tasks

module Handlers =

    let indexView =
        html [] [
            head [] [
                title [] [ str "Giraffe Example" ]
            ]
            body [] [
                h1 [] [ str "I |> F#" ]
                p [ _class "some-css-class"; _id "someId" ] [
                    str "Hello World from the Giraffe View Engine"
                ]
            ]
        ]

    let sayHelloNameHandler (name:string) : HttpHandler =
        fun (next:HttpFunc) (ctx:HttpContext) ->
            task {
                let msg = $"Hello {name}, how are you?"
                return! json {| Response = msg |} next ctx
            }

    let apiRoutes =
        choose [
            route "" >=> json {| Response = "Hello world!!" |}
            routef "/%s" sayHelloNameHandler
        ]

    let webApp =
        choose [
            GET >=> choose [
                route "/" >=> htmlView indexView
                subRoute "/api" apiRoutes
            ]
            RequestErrors.NOT_FOUND "Not Found"
        ]

type Startup() =

    // This method gets called by the runtime. Use this method to add services to th\
e container.
    // For more information on how to configure your application, visit https://go.m\
icrosoft.com/fwlink/?LinkID=398940
    member _.ConfigureServices(services: IServiceCollection) =
        services.AddGiraffe() |> ignore

    // This method gets called by the runtime. Use this method to configure the HTTP\
 request pipeline.
    member _.Configure(app: IApplicationBuilder, env: IWebHostEnvironment) =
        if env.IsDevelopment() then
            app.UseDeveloperExceptionPage() |> ignore

        app.UseGiraffe Handlers.webApp
```

Summary

We have only scratched the surface of what is possible with Giraffe. In this chapter we had an introduction to using Giraffe to build an API and how to use the Giraffe View Engine to create HTML pages. We also learnt about the importance of HttpHandlers in Giraffe routing and we're introduced to a number of the built-in ones Giraffe offers and we have seen that we can create our own handlers.

In the next chapter we will expand the API side of the application.

14 - Creating an API with Giraffe

In this chapter, we'll be creating a simple API with Giraffe. We are going to add new functionality to the project we created in the last chapter.

Getting Started

You will need a tool to run HTTP calls (GET, POST, PUT, and DELETE). I use Postman[33] but any tool including those available in VS Code will work.

Open the code from the last chapter in VS Code.

Our Task

We are going to create a simple API that we can view, create, update and delete Todo items.

Sample Data

Rather than work against a real data store, we are going to create a simple store with a dictionary and use that in our handlers.

Create a new file above Startup.fs called TodoStore.fs and add the following code to it:

```
module GiraffeExample.TodoStore

open System
open System.Collections.Concurrent

type TodoId = Guid

type NewTodo = {
    Description: string
}

type Todo = {
    Id: TodoId
```

[33]https://www.postman.com/

```
    Description: string
    Created: DateTime
    IsCompleted: bool
}

type TodoStore() =
    let data = ConcurrentDictionary<TodoId, Todo>()

    member _.Create todo = data.TryAdd(todo.Id, todo)
    member _.Update todo = data.TryUpdate(todo.Id, todo, data.[todo.Id])
    member _.Delete id = data.TryRemove id
    member _.Get id = data.[id]
    member _.GetAll () = data.ToArray()
```

TodoStore is a simple class type that wraps a concurrent dictionary that we can use to test our API out with. It will not persist between runs.

Add a reference to the to the import declarations in Startup.fs:

```
open GiraffeExample.TodoStore
```

To be able to use the TodoStore, we need to make a change to the configureServices function in Startup.fs:

```
let configureServices (services : IServiceCollection) =
    services
        .AddGiraffe()
        .AddSingleton<TodoStore>(TodoStore()) |> ignore
```

If you're thinking that this looks like dependency injection, you would be correct; We are using the one provided by ASP.NET Core. We add the TodoStore as a singleton as we only want one instance to exist.

Routes

We saw in the last chapter that Giraffe uses individual route handlers, so we need to think about how to add our new routes. The routes we need to add are:

```
GET /api/todo // Get a list of todos
GET /api/todo/id // Get one todo
POST /api/todo // Create a todo
PUT /api/todo/id // Update a todo
DELETE /api/todo/id // Delete a todo
```

Let's create a new handler with the correct HTTP verbs just above the webApp function:

```
let apiTodoRoutes : HttpHandler =
    choose [
        GET >=> choose [
            routef "/%O" viewTodoHandler
            route "" >=> viewTodosHandler
        ]
        POST >=> route "" >=> createTodoHandler
        PUT >=> routef "/%O" updateTodoHandler
        DELETE >=> routef "/%O" deleteTodoHandler
    ]
```

We will create the missing handlers after we have plugged the new handler into our webApp routing handler as a subroute:

```
let webApp =
    choose [
        subRoute "/api/todo" Todos.apiTodoRoutes
        GET >=> choose [
            route "/" >=> htmlView indexView
            subRoute "/api" apiRoutes
        ]
        RequestErrors.NOT_FOUND "Not Found"
    ]
```

There are lots of ways of arranging the routes. It's up to you to find an efficient approach. I like this style with all of the route strings in one function. This makes them easy to change.

Next we have to implement the new handlers.

Handlers

Rather than create the new handlers in Startup.fs, we are going to create them in a new file. We will also move the apiTodoRoutes handler there as well.

Create a new file between Startup.fs and TodoStore.fs called Todos.fs. Copy the following code into the new file:

```
namespace GiraffeExample

open System
open System.Collections.Generic
open Microsoft.AspNetCore.Http
open Giraffe
open FSharp.Control.Tasks
open GiraffeExample.TodoStore

module Todos =
```

Move (Cut & Paste) the apiTodoRoutes handler function to the new Todos module.

You will need to fix the error in the webApp binding by adding 'Todos.' to apiTodoRoutes:

```
subRoute "/todo" Todos.apiTodoRoutes
```

Now we can concentrate on adding the handlers for our new routes to the Todos module above the apiTodoRoutes function. We'll start with the two Get requests:

```
let viewTodosHandler =
    fun (next : HttpFunc) (ctx : HttpContext) ->
        task {
            let store = ctx.GetService<TodoStore>()
            let todos = store.GetAll()
            return! json todos next ctx
        }

let viewTodoHandler (id:Guid) =
    fun (next : HttpFunc) (ctx : HttpContext) ->
        task {
            let store = ctx.GetService<TodoStore>()
            let todo = store.Get(id)
            return! json todo next ctx
        }
```

We are using the HttpContext (ctx) to gain access to the TodoStore instance we set up earlier.

Let's add the handlers for Put and Post:

```
let createTodoHandler =
    fun (next : HttpFunc) (ctx : HttpContext) ->
        task {
            let! newTodo = ctx.BindJsonAsync<NewTodo>()
            let store = ctx.GetService<TodoStore>()
            let created =
                store.Create({
                    Id = Guid.NewGuid()
                    Description = newTodo.Description
                    Created = DateTime.UtcNow
                    IsCompleted = false })
            return! json created next ctx
        }

let updateTodoHandler (id:Guid) =
    fun (next : HttpFunc) (ctx : HttpContext) ->
        task {
            let! todo = ctx.BindJsonAsync<Todo>()
            let store = ctx.GetService<TodoStore>()
            let created = store.Update(todo)
            return! json created next ctx
        }
```

The most interesting thing here is that we use a built-in Giraffe function to gain strongly-typed access to the request body passed into the handler via the HttpContext.

Finally, we handle the Delete route:

```
let deleteTodoHandler (id:Guid) =
    fun (next : HttpFunc) (ctx : HttpContext) ->
        task {
            let store = ctx.GetService<TodoStore>()
            let existing = store.Get(id)
            let deleted = store.Delete(KeyValuePair<TodoId, Todo>(id, existing))
            return! json deleted next ctx
        }
```

We should now be able to use the new Todo API.

Using the API

Run the app and use a tool like Postman to work with the API.

To get a list of all Todos, we call 'GET /api/todo'. This should return an empty json array because we don't have any todos in our store currently.

We create a Todo by calling 'PUT /api/todo' with a json request body like this:

```
{
    "Description": "Finish blog post"
}
```

You will receive a response of true. If you now call the list again, you will receive a JSON response like this:

```
[
    {
        "key": "ff5a1d35-4573-463d-b9fa-6402202ab411",
        "value": {
            "id": "ff5a1d35-4573-463d-b9fa-6402202ab411",
            "description": "Finish blog post",
            "created": "2021-03-12T13:47:39.3564455Z",
            "isCompleted": false
        }
    }
]
```

I'll leave the other routes for you to investigate.

Summary

We have only scratched the surface of what is possible with Giraffe for creating APIs such as content negotiation. I highly recommend reading the Giraffe documentation[34] to get a full picture of what is possible.

In the final chapter of the book, we will dive deeper into HTML views with the Giraffe View Engine.

[34]https://github.com/giraffe-fsharp/Giraffe/blob/master/DOCUMENTATION.md

15 - Creating Web Pages with Giraffe

In the last chapter, we created a simple API for managing a Todo list. In this post, we are going to start our journey into HTML views with the Giraffe View Engine.

If you haven't already done so, read the previous two chapters on Giraffe.

Getting Started

We are going to continue to make changes to the project we updated in the last chapter.

Our Task

We are going to create a simple HTML view of a Todo list and populate it with dummy data from the server.

Rather than rely on my HTML/CSS skills, we are going to start with a pre-built sample: The ToDo list example from w3schools:

https://www.w3schools.com/howto/howto_js_todolist.asp

Configuration

Add a new folder to the project called WebRoot and add a couple of files to the folder: main.js and main.css.

Open the '.fsproj' file and add the following snippet:

```xml
<ItemGroup>
  <Content Include="WebRoot\**\*">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
  </Content>
</ItemGroup>
```

The pattern used allows you to add subfolders but if you do that, you must remember to fix your paths in the views.

We have to tell the app to serve static files if requested. We add some new configuration to the Configure function in Startup.fs:

```
member _.Configure(app: IApplicationBuilder, env: IWebHostEnvironment) =
    if env.IsDevelopment() then
        app.UseDeveloperExceptionPage() |> ignore
    app.UseStaticFiles() |> ignore
    app.UseGiraffe Handlers.webApp
```

The final step to enabling these files to be used is to edit the createHostBuilder function in Program.fs to tell ASP.NET Core to use the new folder to serve static files like JavaScript, CSS, and images:

```
let createHostBuilder args =
    let contentRoot = Directory.GetCurrentDirectory()
    let webRoot = Path.Combine(contentRoot, "WebRoot")
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(fun webBuilder ->
            webBuilder
                .UseStartup<Startup>()
                .UseWebRoot(webRoot) |> ignore
        )
```

We have added a couple of lines to tell the code where the WebRoot folder is and have passed that path to a built-in helper function called UseWebRoot.

That's the end of the configuration but we need to copy the code from the w3schools site for the CSS and JavaScript into our files.

First the CSS:

```
/* Include the padding and border in an element's total width and height */
* {
  box-sizing: border-box;
}

/* Remove margins and padding from the list */
ul {
  margin: 0;
  padding: 0;
}

/* Style the list items */
ul li {
  cursor: pointer;
  position: relative;
  padding: 12px 8px 12px 40px;
  background: #eee;
  font-size: 18px;
  transition: 0.2s;

  /* make the list items unselectable */
  -webkit-user-select: none;
  -moz-user-select: none;
  -ms-user-select: none;
  user-select: none;
}

/* Set all odd list items to a different color (zebra-stripes) */
ul li:nth-child(odd) {
  background: #f9f9f9;
}

/* Darker background-color on hover */
ul li:hover {
  background: #ddd;
}

/* When clicked on, add a background color and strike out text */
ul li.checked {
  background: #888;
  color: #fff;
  text-decoration: line-through;
}

/* Add a "checked" mark when clicked on */
ul li.checked::before {
  content: '';
  position: absolute;
  border-color: #fff;
  border-style: solid;
  border-width: 0 2px 2px 0;
  top: 10px;
  left: 16px;
  transform: rotate(45deg);
  height: 15px;
  width: 7px;
}

/* Style the close button */
.close {
  position: absolute;
  right: 0;
  top: 0;
  padding: 12px 16px 12px 16px;
}

.close:hover {
  background-color: #f44336;
  color: white;
}

/* Style the header */
.header {
  background-color: #f44336;
  padding: 30px 40px;
  color: white;
  text-align: center;
}

/* Clear floats after the header */
.header:after {
  content: "";
  display: table;
  clear: both;
}

/* Style the input */
input {
  margin: 0;
  border: none;
  border-radius: 0;
  width: 75%;
  padding: 10px;
  float: left;
  font-size: 16px;
}

/* Style the "Add" button */
.addBtn {
  padding: 10px;
  width: 25%;
  background: #d9d9d9;
  color: #555;
  float: left;
  text-align: center;
  font-size: 16px;
  cursor: pointer;
  transition: 0.3s;
  border-radius: 0;
}

.addBtn:hover {
  background-color: #bbb;
}
```

Now the JavaScript:

```
// Create a "close" button and append it to each list item
var myNodelist = document.getElementsByTagName("LI");
var i;
for (i = 0; i < myNodelist.length; i++) {
  var span = document.createElement("SPAN");
  var txt = document.createTextNode("\u00D7");
  span.className = "close";
  span.appendChild(txt);
  myNodelist[i].appendChild(span);
}

// Click on a close button to hide the current list item
var close = document.getElementsByClassName("close");
var i;
for (i = 0; i < close.length; i++) {
  close[i].onclick = function() {
    var div = this.parentElement;
    div.style.display = "none";
  }
}

// Add a "checked" symbol when clicking on a list item
var list = document.querySelector('ul');
list.addEventListener('click', function(ev) {
  if (ev.target.tagName === 'LI') {
    ev.target.classList.toggle('checked');
  }
}, false);
```

```javascript
// Create a new list item when clicking on the "Add" button
function newElement() {
  var li = document.createElement("li");
  var inputValue = document.getElementById("myInput").value;
  var t = document.createTextNode(inputValue);
  li.appendChild(t);
  if (inputValue === '') {
    alert("You must write something!");
  } else {
    document.getElementById("myUL").appendChild(li);
  }
  document.getElementById("myInput").value = "";

  var span = document.createElement("SPAN");
  var txt = document.createTextNode("\u00D7");
  span.className = "close";
  span.appendChild(txt);
  li.appendChild(span);

  for (i = 0; i < close.length; i++) {
    close[i].onclick = function() {
      var div = this.parentElement;
      div.style.display = "none";
    }
  }
}
```

Converting HTML To Giraffe View Engine

We are going to create a master page and pass the title and content into it.

Let’s start with a basic HTML page:

```html
<html>
    <head>
        <title>My Title</title>
        <link rel="stylesheet" href="main.css" />
    </head>
    <body />
</html>
```

We will create the function above indexView in the Handlers module in Startup.fs. If we convert the HTML to Giraffe View Engine format, we get:

```
// string -> XmlNode list -> XmlNode
let createMasterPage msg content =
    html [] [
        head [] [
            title [] [ str msg ]
            link [ _rel "stylesheet"; _href "main.css" ]
        ]
        body [] content
    ]
```

Most tags have two lists, one for styling and one for content. Some tags, like *input*, only take the style list.

To create the original HTML view using our new master page, we would change indexView to:

```
let indexView =
    [
        h1 [] [ str "I |> F#" ]
        p [ _class "some-css-class"; _id "someId" ] [
            str "Hello World from the Giraffe View Engine"
        ]
    ]
    |> createMasterPage "Giraffe Example"
```

This approach is very different to most view engines which rely on search and replace but are primarily still HTML. The primary advantage of the Giraffe View Engine approach is that you get full type safety when creating and generating views and can use the full power of the F# language.

Run the website to prove that you haven't broken anything.

The Todo HTML from the w3schools source below has to be converted into Giraffe View Engine format:

```
<div id="myDIV" class="header">
  <h2>My To Do List</h2>
  <input type="text" id="myInput" placeholder="Title...">
  <span onclick="newElement()" class="addBtn">Add</span>
</div>

<ul id="myUL">
  <li>Hit the gym</li>
  <li class="checked">Pay bills</li>
  <li>Meet George</li>
  <li>Buy eggs</li>
  <li>Read a book</li>
  <li>Organise office</li>
</ul>
```

Create a new function below the createMasterPage function to generate our todoView value binding:

```
let todoView =
    [
        div [ _id "myDIV"; _class "header" ] [
            h2 [] [ str "My To Do List" ]
            input [ _type "text"; _id "myInput"; _placeholder "Title..." ]
            span [ _class "addBtn"; _onclick "newElement()" ] [ str "Add" ]
        ]
        ul [ _id "myUL" ] [
            li [] [ str "Hit the gym" ]
            li [ _class "checked" ] [ str "Pay bills" ]
            li [] [ str "Meet George" ]
            li [] [ str "Buy eggs" ]
            li [] [ str "Read a book" ]
            li [ _class "checked" ] [ str "Organise office" ]
        ]
        script [ _src "main.js"; _type "text/javascript" ] []
    ]
    |> createMasterPage "My ToDo App"
```

We are nearly ready to run; We only need to tell the router about our new view handler. Change the root '/' route in webApp to:

```
GET >=> route "/" >=> htmlView todoView
```

If you now run the app using dotnet run in the terminal. Point your browser at https://localhost:5001 and you will see the Todo app running.

Next Stage

Rather than hard code the list of Todos into the view, we can load it in on the fly by creating a list of items. Create the following above the todoView:

```
let todoList = [
    { Id = Guid.NewGuid(); Description = "Hit the gym"; Created = DateTime.UtcNow; I\
sCompleted = false }
    { Id = Guid.NewGuid(); Description = "Pay bills"; Created = DateTime.UtcNow; IsC\
ompleted = true }
    { Id = Guid.NewGuid(); Description = "Meet George"; Created = DateTime.UtcNow; I\
sCompleted = false }
    { Id = Guid.NewGuid(); Description = "Buy eggs"; Created = DateTime.UtcNow; IsCo\
mpleted = false }
    { Id = Guid.NewGuid(); Description = "Read a book"; Created = DateTime.UtcNow; I\
sCompleted = true }
    { Id = Guid.NewGuid(); Description = "Read Essential Functional-First F#"; Creat\
ed = DateTime.UtcNow; IsCompleted = false }
]
```

There's a lot of duplicate code, so let's create a helper function to simplify things:

```
let create description isCompleted =
    {
        Id = Guid.NewGuid()
        Description = description
        Created = DateTime.UtcNow
        IsCompleted = isCompleted
    }

let todoList =
    [
        create "Hit the gym" false
        create "Pay bills" true
        create "Meet George" false
        create "Buy eggs" false
        create "Read a book" true
        create "Read Essential Functional-First F#" false
    ]
```

We need to create a partial view, which is simple helper function to style each item in the Todo list. You should write this function above the todoView code:

```
// Todo -> XmlNode
let showListItem (todo:Todo) =
    let style = if todo.IsCompleted then [ _class "checked" ] else []
    li style [ str todo.Description ]
```

Now we can use the showListItem partial view function in the todoView and we can pass the Todo items in as an input parameter:

```
let todoView items =
    [
        div [ _id "myDIV"; _class "header" ] [
            h2 [] [ str "My ToDo List" ]
            input [ _type "text"; _id "myInput"; _placeholder "Title..." ]
            span [ _class "addBtn"; _onclick "newElement()" ] [ str "Add" ]
        ]
        ul [ _id "myUL" ] [
            for todo in items do
                showListItem todo
        ]
        script [ _src "main.js"; _type "text/javascript" ] []
    ]
    |> createMasterPage "My ToDo App"
```

We need to change the root route as we are now passing the list of Todos into the todoView function:

```
GET >=> route "/" >=> htmlView (todoView todoList)
```

Run the app to see the items displayed on the HTML page.

What we also have is view code where it probably shouldn't be, so let's have a go at tidying up a little.

Tidying Up

We have a master page and a view of Todos with supporting code that need moving. Let's move things around and see what we've broken by doing so.

Let's start by creating a new file called Shared.fs above the existing files and add the createMaster-Page function to it. Fixing this code means adding an import declaration for the Giraffe ViewEngine as well:

```
namespace GiraffeExample

module SharedViews =

    open Giraffe.ViewEngine

    let createMasterPage msg content =
        html [] [
            head [] [
                title [] [ str msg ]
                link [ _rel "stylesheet"; _href "css/main.css" ]
            ]
            body [] content
        ]
```

It is recommended that you limit the scope of the import declaration to the module containing the view functions.

We can delete the indexView from Startup.fs as it is no longer used.

We need to fix the todoView function as it can't find the master page that we have just moved by adding the name of the new module:

```
|> SharedViews.createMasterPage "My ToDo App"
```

Create a new module inside the Todos module in Todos.fs called Views, add an import declaration for the Giraffe View Engine and move the two todo view functions from Startup.fs into it:

```
module Todos =

    module Views =

        open Giraffe.ViewEngine

        let showListItem (todo:Todo) =
            let style = if todo.IsCompleted then [ _class "checked" ] else []
            li style [ str todo.Description ]

        let todoView items =
            [
                div [ _id "myDIV"; _class "header" ] [
                    h2 [] [ str "My ToDo List" ]
                    input [ _type "text"; _id "myInput"; _placeholder "Title..." ]
                    span [ _class "addBtn"; _onclick "newElement()" ] [ str "Add" ]
                ]
                ul [ _id "myUL" ] [
                    for todo in items do
                        showListItem todo
                ]
                script [ _src "main.js"; _type "text/javascript" ] []
            ]
            |> SharedViews.createMasterPage "My ToDo App"
```

The final fix is to let the webApp function know where to find the todoView function:

```
let webApp =
    choose [
        subRoute "/api/todo" Todos.apiTodoRoutes
        GET >=> choose [
            route "/" >=> htmlView (Todos.Views.todoView todoList)
            subRoute "/api" apiRoutes
        ]
        RequestErrors.NOT_FOUND "Not Found"
    ]
```

If you build the website now, it should be ready to run.

Run the website and check that the Todos page displays correctly.

I highly recommend reading the Giraffe View Engine documentation[35] to find out about some of the other features available.

If you like the Giraffe View Engine but don't like writing JavaScript, you should investigate the SAFE Stack[36] where **everything** is written in F#.

Summary

We have only scratched the surface of what is possible with Giraffe and the Giraffe View Engine for creating web pages and APIs.

That concludes our journey into functional-first F# and Giraffe. I hope that I have given you enough of a taste of what F# can offer to encourage you to want to take it further.

[35]https://github.com/giraffe-fsharp/Giraffe.ViewEngine
[36]https://safe-stack.github.io/

Summary

Congratulations on reaching the end of this book. I hope that it has been a fun journey and I have conveyed to you some of joy that I get from working every day on F# codebases. Even if you don't get the chance to work in F#, many the ideas and practices in this book will still impact the way you think about and write code.

We have covered a lot in just over 160 pages! During the course of reading this book, you have been introduced to the essentials of functional-first F#:

- F# Interactive (FSI)
- Algebraic Type System
  - Records
  - Discriminated Unions
  - Tuples
- Pattern Matching
  - Active Patterns
  - Guard Clauses
- Let bindings
- Immutability
- Functions
  - Pure
  - Higher Order
  - Currying
  - Partial Application
  - Pipelining
  - Signatures
  - Composition
- Unit
- Effects
  - Option
  - Result
  - Async
  - Task
- Computation Expressions
- Collections
  - Lists

    - Sequences
    - Arrays
    - Sets
- Recursion
- Object Programming
    - Class types
    - Interfaces
    - Object Expressions
- Exceptions
- Unit Tests
- Namespaces and Modules

You've also started the journey towards writing APIs and websites in Giraffe.

What makes F# such a wonderful language to use is not just the individual features but how they work together; It's the epitome of Aristotle's oft-quoted words:

> The whole is greater than the sum of its parts.

F# Software Foundation

Thank you for purchasing this book. All of the author's royalties will be going to the F# Software Foundation to support their efforts in promoting the F# language and community to the world.

If you haven't already joined, you should join the F# Software Foundation[37]; It's free. By joining, you will be able to access the dedicated Slack channel where there are excellent tracks for beginners and beyond.

Resources

There are plenty of other high quality resources available for F#. These are some that I heartily recommend:

Books

- Get Programming With F#[38]
- Stylish F#[39]
- Domain Modelling Made Functional[40]

[37]https://foundation.fsharp.org/join
[38]https://www.manning.com/books/get-programming-with-f-sharp
[39]https://www.apress.com/us/book/9781484239995
[40]https://www.pragprog.com/titles/swdddf/domain-modeling-made-functional/

Websites

- F# Docs[41]
- F# Software Foundation[42]
- F# For Fun and Profit[43]
- F# Weekly[44]
- Compositional-It Blog[45]

Courses

- F# From the Ground Up[46]

And Finally

Follow me on Twitter[47] and keep an eye on my Company's blog[48] for further F# posts along with lots of other interesting stuff on Domain-Driven Design, Machine Learning, and many other things.

[41]https://fsharp.github.io/fsharp-core-docs
[42]https://fsharp.org/
[43]https://fsharpforfunandprofit.com/
[44]https://sergeytihon.com/category/f-weekly/
[45]https://www.compositional-it.com/news-blog/
[46]https://www.udemy.com/course/fsharp-from-the-ground-up/
[47]https://twitter.com/ijrussell
[48]https://www.softwarepark.cc/blog

Appendix

Creating Solutions and Projects in VS Code

We are going to create a new .NET Solution containing an F# console project and an XUnit test project using the dotnet CLI.

Open VS Code in a new folder.

Open a new Terminal window. The shortcut for this is CTRL+SHIFT+'.

Create a new file in the folder. It doesn't matter what it is called, so use 'setup.txt'.

Copy the following script into the file:

```
dotnet new sln -o MySolution
cd MySolution
mkdir src
dotnet new console -lang F# -o src/MyProject
dotnet sln add src/MyProject/MyProject.fsproj
mkdir tests
dotnet new xunit -lang F# -o tests/MyProjectTests
dotnet sln add tests/MyProjectTests/MyProjectTests.fsproj
cd tests/MyProjectTests
dotnet add reference ../../src/MyProject/MyProject.fsproj
dotnet add package FsUnit
dotnet add package FsUnit.XUnit
dotnet build
dotnet test
```

This script will create a new solution called MySolution, two folders called src and tests, a console app called MyProject in the src folder, and a test project called MyProjectTests in the tests folder. Change the names to suit. In VS Code, CTRL+F2 will allow you to edit all of the instances of the selected word at the same time.

Select all of the script text.

To run the script in the Terminal, you can do either of the following:

- Choose the Terminal menu item and then select 'Run Selected Text'.
- Press CTRL+SHIFT+P to open the Command Palette and then type 'TRSTAT'. Select the '**T**erminal: **R**un **S**elected **T**ext in **A**ctive **T**erminal' item.

The script will now execute in the Terminal.

You can delete the file with the script in if you want as you no longer need it.