practical functional refactoring tips for the imperative world - part 2: effectful f#

introduction

This article covers a practical to follow refactoring workflow which helps transform difficult to manage legacy code into a more functional, easier to maintain version of itself. It is a follow up to an article I published last quarter which covered specifically the pure components of any program. You can find it here: practical functional refactoring tips for the imperative world. Every computer program consists of a mixture of two types of code:

- **pure code** - no side effects, code executed multiple times with the same input always yields the same results (think Excel spreadsheet)
- **effectful/impure code** - code causing side effects: database/file system/network interactions but also things as trivial as telling the time (think distributed system where many machines are working together)

This article focuses on the latter: the effectful code. It should be a bit closer to real life compared to the last article in the "practical functional refactoring tips" series and serve as a reference for the most common day-to-day tasks most programmers handle. The aim is to embrace side effects as something that just needs to happen, but also to keep them manageable and explicit, so that the code is relatively easy to maintain and extend.

Finally, the article should also serve as a good starting point for those wanting to experiment with adopting F# but are not sure where to start.

how did we get here

Most of 2021 we spent getting familiar with functional programming and with F# by adapting it for the pure responsibilities of our code. That gave everybody the much needed time to get to know the language, the ecosystem, the C#-F# interop tricks and gotchas. The basic building blocks of any new language.

Now that those aspects are no longer a mystery, we are ready for the more tricky effectful code. Why tricky? Everything boils down to how to work with asynchrous code...

asynchronous code

The trick to understanding asynchronous programming is to accept what it is/is not about:

- **what it is not about**:
  - concurrency - when multiple computations execute in overlapping time periods
  - parallelism - when multiple computations or several parts of a single computation run at exactly the same time
- **what it is about**:
  - asynchrony - when one or more computations can execute separately from the main program flow (asynchrony quite literally means "not-at-the-same-time")

source - Async programming in F#

asynchrony in f#

Let's start with the reference material and terminology:

- `Async<T>` type represents composable asynchronous operations

- `Async` module contains functions which:

  - schedule
  - compose
  - transform

  asynchronous operations like:

  - `Async.RunSynchronously`
  - `Async.Start`
  - `Async.StartChild`
  - `Async.Catch`
  - `Async.AwaitTask`

- `async { }` - computation expression/asynchronous expression. All expressions of such form are of type `Async<T>` for some `T`. *source*

- `expr := async { aexpr }` - complete syntax for asynchronous expressions. Selected `aexpr` examples:

  - `do!` - execute async (`Async<unit>`).
  - `let!` - execute and bind async (`Async<T>`).
  - `return! expr` - tailcall to async
  - `return expr` - return result of async expression

But here is where it gets interesting and where most implementation challenges are going to surface:

- C# - F# interop (`Async` vs `Task`)
- exceptions
  - C# exception types are supported
- cancellations
  - C#'s `CancellationTokenSource` and `CancellationToken` are both supported
  - cancellation tokens are implicitly propagated through the execution of nested asynchronous operations
  - cancellation tokens are provided at the entry point to the execution of an asynchronous computation, e.g.: `Async.RunSynchronously`, `Async.StartImmediate`, `Async.Start`
  - cancelling nested asynchronous expressions: cancellation tokens are passed implicitly to nested asynchronous expressions, but depending on how asynchronous work gets started, cancellations are handled differently in the nested operations, e.g.:
    - Async.Start - If the parent computation is canceled, no child computations are canceled.
    - Async.StartChild - If the parent computation is canceled, the child computation is also canceled.

mercury-functional

In the article I published previously here: practical functional refactoring tips for the imperative world, we went through refactoring the pure responsibilities out of C# mercury-legacy into mercury-pure-functional - a C#-F# hybrid where the pure code was written in F# and the effectful code was written in C#.

This time, we are going to need a new repository: mercury-functional written exclusively in F# and based on mercury-pure-functional.

Previously, we isolated all pure responsibilities and divided them between 4 F# modules:

- Models
- InputValidation
- Mappers
- BusinessLogic

Leaving the Host written in C#. This time, we will be focusing on the Host project.

mercury code

Compared to `mercury-pure-functional`, both repositories are quite similar with one exception: the Host project. The project contains 3 functions which represent interesting differences between both implementations and we'll take a close look at all three. Let's start with the `main`/`Main` functions (the entry point to the program):

main - C#

```
public static async Task Main(string[] args)
{
    Log("function is starting...");

    var apiUrlFormat = args[0];
```

```
    var domain = args[1];

    FSharpOption<Models.WhoisResponse> response = await GetWhoisResponse(
        apiUrlFormat,
        domain);

    Log(response.ToString());
    Log("function execution finished");
}
```

main - F#

```
[<EntryPoint>]
let main argv =
    log "function is starting..."

    let apiUrlFormat = argv.[0]

    let domain = argv.[1]

    let job =
        async { return! getWhoisResponse apiUrlFormat domain }

    let response = Async.RunSynchronously(job)

    printfn $"{response.ToString()}"

    log "function execution finished"

    0
```

differences

- In the C# version, the `response` object is a result of an asynchronous function (`GetWhoisResponse`) being started and `await`ed.
- In the F# version, we have slightly more explicitly defined options of how the asynchronous `job` can get executed:
    - very much like a traditional C# `Task`, `job` only gets created and not immediately invoked
    - `response` is a result of `job` getting executed synchronously (no real benefits running it synchronously in our case), but other options are also available, e.g. `Async.StartChild` (for asynchronous execution).
- `!` notation allows us to nearly seamlessly introduce `Async` type into the code without too many changes. The key difference beeing the `async` vs expression results:
    - `let! pat = expr in aexpr` - execute & bind async. Example:

    ```
    // apiResponse is of type HttpResponseMessage
    let! apiResponse = client.GetAsync(apiUrl, cancellationToken) |>
    ```

```
Async.AwaitTask
```

- `let pat = expr in aexpr` - execute & bind expression

```
// apiResponse is of type Async<HttpResponseMessage>
let apiResponse = client.GetAsync(apiUrl, cancellationToken) |>
Async.AwaitTask
```

  source - chapter 2, An Overview of F# Asynchronous Programming
- It is possible to pass in a `CancellationToken` to `Async.RunSynchronously` and let it trickle down the execution chain (that is done implicitly) but in our case, `getWhoisResponse` has its own `CancellationTokenSource`.

log

Both functions are nearly identical, synchronous and as a result, comparing them is beyond the scope of this article.

getWhoisResponse - C#

This is the busiest function of the project, let's take a close look at what's different between both implementations:

```
private static async Task<FSharpOption<Models.WhoisResponse>>
GetWhoisResponse(
    string apiUrlFormat,
    string domain)
{
    FSharpOption<Models.WhoisResponse> response = null;

    InputValidation.whoisInputValidation(apiUrlFormat, domain);

    var apiUrl = string.Format(apiUrlFormat, domain);

    var cancellationTokenSource = new
CancellationTokenSource(TimeSpan.FromSeconds(3));

    var cancellationToken = cancellationTokenSource.Token;

    var client = new HttpClient();

    var apiResponse = await client.GetAsync(
        apiUrl,
        cancellationToken);

    if (apiResponse.IsSuccessStatusCode)
    {
        var serializer = new XmlSerializer(typeof(Models.WhoisRecord));
```

```
        await using Stream reader = await
apiResponse.Content.ReadAsStreamAsync(cancellationToken);

        response = Mappers.toWhoisResponse(
            DateTime.Now,
            domain,
            (Models.WhoisRecord)serializer.Deserialize(reader));
    }

    return response;
}
```

getWhoisResponse - F#

```
let getWhoisResponse (apiUrlFormat: string) (domain: string) :
Async<WhoisResponse option> =
    InputValidation.whoisInputValidation apiUrlFormat domain

    let apiUrl = String.Format(apiUrlFormat, domain)

    let cancellationTokenSource =
        new CancellationTokenSource(TimeSpan.FromSeconds(3.0))

    let cancellationToken = cancellationTokenSource.Token

    let client = new HttpClient()

    async {
        let! apiResponse =
            client.GetAsync(apiUrl, cancellationToken)
            |> Async.AwaitTask

        if apiResponse.IsSuccessStatusCode then
            let serializer = XmlSerializer(typeof<WhoisRecord>)

            let! stream =
                apiResponse.Content.ReadAsStreamAsync(cancellationToken)
                |> Async.AwaitTask

            let whoisRecord =
                serializer.Deserialize(stream) :?> WhoisRecord

            return Mappers.toWhoisResponse DateTime.Now domain whoisRecord
        else
            return Option.None
    }
```

differences

- code spanning from input validation to `HttpClient` creation is virtually identical in both scenarios, but what deserves special attention is that the `CancellationToken`\`CancellationTokenSource` classes are usable both from C# and F#. What follows is where the more notable differences start.

- while the C# version does not need the `async` code block, the F# counterpart does. That allows for more visually pleasing, easier to follow syntax inside that block on the F# side (leveraging the `!` notation). It is not uncommon for asynchronous F# functions to only consist of one `async` block and nothing else outside it.

- while the C# return type is `Task<FSharpOption<Models.WhoisResponse>>`, the F# return type is `Async<WhoisResponse option>`. C# relies on the `Task` type to describe asynchrony while F# relies on the `Async` type. Conversions and interop between both is going to be covered in following bullet points. F#'s `Async` return type is a result of using the `async` block and what should be noted is that while on the C# side, we start and await the asynchronous work, on the F# side, we only specify how it should be executed. The asynchronous expression is not actually started in the `getWhoisResponse` function. It is the caller's responsibility to execute the returned `Async` object and in our case the caller does it like so:

  ```
  let job =
      async { return! getWhoisResponse apiUrlFormat domain }

  let response = Async.RunSynchronously(job)
  ```

  That is one of the key differences between C# and F# approaches to asynchrony.

- `HttpClient`: it is important to note that while the reusable (C#/F#) `HttpClient` methods are executed the same way in both languages, there are two critical differences:

  - F#'s `let!` execution and binding of the asynchronous expression vs C#'s `await`. Both result in creating an object of type `HttpResponseMessage`.
  - while the C# code (when un`awaited`) returns a `Task`, the F# code needs to translate that `Task` into a type it depends on for asynchrony (`Async`). To do that, we simply pipe the returned `Task` into the `Async`'s `AwaitTask` which translates `Task` to `Async` (on which we can use the `!` notation).

  The same we can observe in the lines using `ReadAsStreamAsync` and that is basically how C# - F# interop works: `Task`s get translated to `Async` objects. More details and examples in a very comprehensive paper F# creators produced: [The F# Asynchronous Programming Model].

- Casting. While the C# part's casting looks familiar:

  ```
  (Models.WhoisRecord)serializer.Deserialize(reader)
  ```

  its F# counterpart is slightly more exotic:

  ```
  serializer.Deserialize(stream) :?> WhoisRecord
  ```

The `:?>` operator performs a dynamic downcast, which means that the success of the cast is determined at run time. source - downcasting

conclusion

Those are the most notable differences between asynchrony in both languages. F# does not introduce any revolutionary concepts in comparison to C#, the differences are more subtle. What the engineers gain, though, is an asynchrnous programming model that is more difficult to use **incorrectly**, e.g. asynchrnonous work cannot be fired-and-forgotten by mistake - if fire-and-forget is the intention, such work needs to be kicked off explicitly and intentionally (as opposed to C#'s `await` omission, which is a mistake in most scenarios). This article gently introduces fundamental concepts of asynchronous programming in F# and should serve as a good starting point for introducing F# and experimenting with it. In upcoming articles we will share more low level details, common pitfalls of asynchronous code (both in C# and in F#) and how to avoid them.

The overall objective of the "practical functional refactoring tips" series of articles is not necessarily to introduce F#, but rather to expose the reader to a more functional way of thinking and breaking down programming problems. Doing so often tends to lead to producing more maintainable, concise, easier to follow code. Having mastered the generally applicable basics of functional thinking (like the explicit separation of pure and effectful responsibilities), it should not be a big challenge to introduce any language from the large, and constantly growing, functional family of languages like F#, Haskell, Erlang, Scala, etc.

further reading

- Async programming in F#
- The F# Asynchronous Programming Model
- F# Async Guide
- Symbol and operator reference
- Casting and conversions (F#)