csharp-functional-docs Documentation

Release 1.0.0

Eduardo Serrano

Sep 03, 2017

Contents:

1 About the repository **3**
- 1.1 Source Code . . . 3
- 1.2 Documentation . . . 4

2 Maybe monad **7**
- 2.1 Installing . . . 8
- 2.2 How To . . . 8

3 Result monad **11**
- 3.1 Installing . . . 11
- 3.2 Result: How To . . . 11
- 3.3 ResultWithError: How To . . . 12
- 3.4 Result<TValue>: How To . . . 12
- 3.5 Result<TValue,TError>: How To . . . 13

4 HttpResult monad **15**
- 4.1 Installing . . . 15
- 4.2 HttpResult monads: How To . . . 15
- 4.3 IHttpState: How To . . . 16
- 4.4 HttpResult monad is disposable . . . 17

5 Combine methods **19**
- 5.1 Combine methods for Result monads . . . 19
- 5.2 Combine methods for HttpResult monads . . . 20
- 5.3 Examples . . . 20

6 Extensions **23**
- 6.1 NuGet packages . . . 24
- 6.2 Installing . . . 25
- 6.3 Maybe extensions: examples . . . 25
- 6.4 Result and HttpResult extensions: examples . . . 25
- 6.5 Mapping from one monad type to another . . . 27
- 6.6 Putting it all together: Railway Oriented Programming . . . 27

7 More resources **33**

The csharp-functional repository started as a learning experience about functional programming concepts. The main trigger was a Pluralsight video from Vladimir Khorikov named Applying Functional Principles in C#.

After watching the video I immediately tried to apply the concepts in one of my pet projects and I found that I wanted a bit more than the functionality described in the Pluralsight course. In the course the Result type that is described is capable of holding or not a value, so you have: Result or Result<T>. In both cases you have an error property which is of type string that you can chose to set to describe what went wrong. I felt that using a string to describe an error was not what I wanted in most cases. So all of this started because I wanted to create a Result monad which could have an error of any type. It turned out to be much more complex than I could have imagined...

I highly advise you to watch that Pluralsight course as well as to read Eric Lippert's series of blog posts on monads.

This documentation aims to provide enough instructions to successfully use the *NuGet packages* as well as understanding the code in the repository. While reading it keep in mind that:

- Although it mentions monads, it's outside of the scope to try to explain what a monad is.
- The *more resources* section contains links that will be helpful to understand what a monad is.
- Since my understanding of monads is limited it might very well be possible that I sometimes use the word incorrectly.
- All code examples are meant to illustrate the usage of the *NuGet packages* and are not meant to reflect real world code practices.
- Whenever I say an ok result/httpresult I mean a result/httpresult that has the IsSuccess property equal to true.
- Whenever I say a fail result/httpresult I mean a result/httpresult that has the IsFailure property equal to true.

CHAPTER 1

About the repository

Source Code

There is only one solution in the csharp-functional repository:

- Source/CSharpFunctional/CSharpFunctional.sln

Structure of the solution

Description	Type	NuGet	Location
The Maybe monad	Source	MaybeMonad NuGet	MaybeMonad
The Result monad	Source	ResultMonad NuGet	ResultMonad
The HttpResult monad	Source	HttpResultMonad NuGet	HttpResultMonad
The Result monad extensions	Source	ResultMonad.Extensions NuGet	ResultMonad.Extensions
The HttpResult monad extensions	Source	HttpResultMonad.Extensions NuGet	HttpResultMonad.Extensions
The Result monad extensions that *transforms them* into HttpResult monad	Source	ResultMonad.Extensions.HttpResultMonad NuGet	ResultMonad.Extensions.HttpResultMonad
The Maybe monad extensions that *transforms them* into Result monad	Source	MaybeMonad.Extensions.ResultMonad NuGet	MaybeMonad.Extensions.ResultMonad
Simple application of HttpResult monad on a class by using it with System.Net.Http.HttpClient	Source	HttpResultMonad.HttpResultOnHttpClient NuGet	HttpResultMonad.HttpResultOnHttpClient
Tests for the Maybe monad	Test	N/A	MaybeMonad.Tests
Tests for the Result monad	Test	N/A	ResultMonad.Tests
Tests for the HttpResult monad	Test	N/A	HttpResultMonad.Tests
Tests for the Result monad extensions	Test	N/A	ResultMonad.Extensions.Tests
Tests for the HttpResult monad extensions	Test	N/A	HttpResultMonad.Extensions.Tests
Tests for the Result monad extensions that *transforms them* into HttpResult monad	Test	N/A	ResultMonad.Extensions.HttpResultMonad.Tests
Tests for the Maybe monad extensions that *transforms them* into Result monad	Test	N/A	MaybeMonad.Extensions.ResultMonad.Tests
Tests for the simple example of applying HttpResult monad on a class by using it with System.Net.Http.HttpClient	Test	N/A	HttpResultMonad.HttpResultOnHttpClient.Tests
Shared code between test projects	Test Library	N/A	Tests.Shared

Building the solution and running tests

This repository adheres to the F5 manifesto so you should be able to clone, open the solution in Visual Studio and build/run tests.

Documentation

The documentation for the repository can be found at the Docs folder.

Read this to understand how the documentation was created and how you can build it.

Building the docs

To build the docs you need:

- Python;
- Sphinx;
- Sphinx read the docs theme.
- Pylint (not required but recommended)

You can install Python from here or if you have chocolatey you can do the following from PowerShell:

```
choco install python
```

Sphinx is a tool that makes it easy to create beautiful documentation. Assuming you have Python already, install Sphinx by executing the following on PowerShell:

```
pip install sphinx sphinx-autobuild
```

The read the docs theme is configured in the conf.py file. To get this theme execute the following on PowerShell:

```
pip install sphinx_rtd_theme
```

For more information about the read the docs theme see its repo.

To install Pylint execute the following from PowerShell:

```
pip install pylint
```

Once you have setup your environment you can build the docs by running the make.bat file. You can also build the docs from Visual Studio Code as explained in the next section.

Note: These build instructions are focused for Windows users. If you are using a different OS then the instructions can't be taken word by word but the same requirements apply. Furthermore there is a makefile available for non Windows users.

Editing the docs with Visual Studio Code

Although you can use your editor of preference to work with reStructuredText files, I have found Visual Studio Code to be very good for this task.

After installing Visual Studio Code I recommend installing the following extensions:

- Code Spell Checker;
- reStructuredText;
- Python.

These will greatly increase your productivity while editing the documentation. For instance:

- The Code Spell Checker will highlight words it does not recognize and then you can use Ctrl+. on those words to correct them.
- The reStructuredText extension will

- Visual Studio Code contains numerous features that will improve your productivity. Something as simple as providing auto-complete suggestions from the words already available on your documentation speeds up your typing a lot.

In the .vscode folder you have 4 configuration files:

- **cSpell.json**: contains configuration for the Code Spell Checker extension. You can add words to this file by using the Ctrl+. shortcut on words that the spell checker does not recognize and chose "Add to project dictionary".
- **settings.json**: contains the configuration for the reStructuredText extensions.
- **tasks.json**: contains 3 tasks for Visual Studio Code:
  - **build-docs**: The default build task. In Visual Studio Code go to "Tasks->Run Task" and select the build-docs task or "Tasks->Run Build Task". If you're using Visual Studio Code from a non Windows OS then you should change the command to execute the makefile instead of the make.bat.
  - **open-docs**: A task to open the docs. In Visual Studio Code go to "Tasks->Run Task" and select the open-docs task. That should open the index.html using your default browser. The documents must have been built first.
  - **build-open-docs**: A task to build and then open the docs. In Visual Studio Code go to "Tasks->Run Task" and select the build-open-docs task. That should build the docs and then open the index.html using your default browser.
- **keybindings.json**: contains some bindings that help you build and open the docs. If you want to use them you have to go to "File->Preferences->Keyboard Shortcuts" then on the top left you will see a message saying "For advanced customizations open and edit the keybindings.json". Select the link under keybindings.json and edit the file to add the contents from the this file. This will add 4 keybindings:
  - **shift+f5** : opens the run task menu;
  - **f6** : runs the build task;
  - **f5** : runs the build-open-docs task;
  - **f10** : runs the open-docs task.

Once you have installed Visual Studio code and the recommended extensions you can edit and build the docs by going to "File->Open Folder" and choosing the "/Docs/source" directory. If you do not open the source folder the build task will fail to run.

CHAPTER 2

Maybe monad

The Maybe monad encapsulates an optional value. An instance of Maybe either has a value of the encapsulated type or it doesn't in which case it is a Maybe<T>.Nothing. This type is meant to be used in cases where your method might or might not return a value. Consider the following:

```
public class UserRepository
{
        public User GetById(int id)
        {
                //some implementation
        }
}

User user = _repository.GetById(id);
```

What happens if the user does not exist? Often a null value is returned. There are at least 2 problems with this:

- The method GetById is not honest because it says it will return a User but in truth it returns a User instance or null.
- The compiler has no way to assist you detecting that the variable user can be null. You are responsible for making sure your code always handles null values properly which might be trivial if you consider just this example but on a large code base it is not.

Applying the Maybe monad to the above example would result in the following:

```
public class UserRepository
{
        public Maybe<User> GetById(int id)
        {
                //some implementation
        }
}

Maybe<User> user = _repository.GetById(id);
```

This fixes both of the previous issues because now:

- The method is honest and anyone using it understands that GetById might or might not return an instance of User.
- The variable user is never null.

Installing

The Maybe monad NuGet package can be found at MaybeMonad NuGet. Installing is performed via NuGet:

```
PM> Install-Package MaybeMonad
```

How To

To create a Maybe instance without value do:

```
var emptyMaybe = Maybe<int>.Nothing;
```

To create a Maybe instance with value do:

```
var maybeWithValue = Maybe.From("some text");
```

Attempting to create a Maybe instance with a null value results in an empty Maybe:

```
var maybeWithNullValue = Maybe.From<string>(null);
var isEqual = maybeWithNullValue.Equals(Maybe<string>.Nothing); //evaluates to true
```

To check if a Maybe instance has or does not have value do:

```
var emptyMaybe = Maybe<int>.Nothing;
var hasValue = emptyMaybe.HasValue;      //evaluates to false
var hasNoValue = emptyMaybe.HasNoValue;  //evaluates to true
```

Accessing the value of an empty Maybe throws an InvalidOperationException exception:

```
var emptyMaybe = Maybe<int>.Nothing;
var value = emptyMaybe.Value; // throws exception
```

The Maybe type as an implicit converter for the type that is encapsulated. This means that you can do:

```
Maybe<int> maybeInt = 2; //the implicit converter transforms the int 2 into Maybe<int>
↪ with a value of 2
```

Or more useful:

```
public class UserRepository
{
        public Maybe<User> GetById(int id)
        {
                User user;
                //db call to query for the user

                if(/*the user was not found*/)
                {
                        return null; //the implicit converter means that what is␣
↪returned is Maybe<User>.Nothing;
```

```
                }

                //the user was found, populate the user variable;
                return user; //the implicit converter means that what is returned is␣
↪Maybe.From(user);
            }
}
```

CHAPTER 3

Result monad

The essence of the Result monad is to augment the outcome of an operation with a success status. There are four variations of the Result monad:

- Result : simply augments the outcome of an operation with a success status.
- ResultWithError<TError> : besides the success status it includes an Error property which must have a value if the success status is false.
- Result<TValue> : besides the success status it includes a Value property which must have a value if the success status is true.
- Result<TValue,TError> : besides the success status it includes a Value property which must have a value if the success status is true and an Error property which must have a value if the success status is false.

All variations of the Result monad contain an IsSuccess and it's inverse IsFailure property that indicate if the operation was successful or not.

Installing

The Result monad NuGet package can be found at ResultMonad NuGet. Installing is performed via NuGet:

```
PM> Install-Package ResultMonad
```

Result: How To

The Result type is meant to be used when the method does not return any value and if it fails you do not care about the details of the failure.

To create a Result instance do:

```
var okResult = Result.Ok();
var failResult = Result.Fail();
```

ResultWithError: How To

The ResultWithError<TError> type is meant to be used when the method does not return any value but if it fails you want information regarding the failure.

To create a ResultWithError instance do:

```
var okResult = ResultWithError.Ok<string>();
var failResult = ResultWithError.Fail("some error info");
```

To access the error do:

```
var failResult = ResultWithError.Fail("some error info");
var error = failResult.Error; // evaluates to "some error info"
```

Accessing the Error property of an ok ResultWithError throws InvalidOperationException exception:

```
var okResult = ResultWithError.Ok<string>();
var error = failResult.Error; //throws exception
```

You can also use the From method to create a ResultWithError. It accepts a predicate function and an error. If the predicate evaluates to true the method returns an ok ResultWithError, if the predicate evaluates to false it returns a fail ResultWithError containing the error:

```
var okResult = ResultWithError.From(()=>true, "some error info");       // creates an␣
↪ok ResultWithError<string>
var failResult = ResultWithError.From(()=>false, "some error info");    // creates a␣
↪fail ResultWithError<string> with "some error info" as the error
```

Result<TValue>: How To

The Result<TValue> type is meant to be used when the method does returns a value but if it fails you do not want information regarding the failure.

To create a Result<TValue> instance do:

```
var okResult = Result.Ok("some value");
var failResult = Result.Fail<string>();
```

To access the value do:

```
var okResult = Result.Ok("some value");
var value = okResult.Value; // evaluates to "some value"
```

Accessing the Value property of a fail Result<TValue> throws InvalidOperationException exception:

```
var okResult = Result.Ok<string>();
var error = okResult.Error; //throws exception
```

You can also use the From method to create a Result<TValue>. It accepts a predicate function and a value. If the predicate evaluates to true the method returns an ok Result<TValue> containing the value, if the predicate evaluates to false it returns a fail Result<TValue>:

```
var okResult = Result.From(()=>true, "some value");     // creates an ok Result
↪<string> with "some value" as the value
var failResult = Result.From(()=>false, "some value");  // creates a fail Result
↪<string>
```

Result<TValue,TError>: How To

The Result<TValue,TError> type is meant to be used when the method does returns a value and if in addition, if it fails, it will return an error.

To create a Result<TValue,TError> instance do:

```
var okResult = Result.Ok<string,int>("some value");
var failResult = Result.Fail<string,int>(0);
```

To access the value do:

```
var okResult = Result.Ok<string,int>("some value");
var value = okResult.Value; // evaluates to "some value"
```

To access the error do:

```
var failResult = Result.Fail<string,int>(0);
var error = failResult.Error; // evaluates to 0
```

Accessing the Value property of a fail Result<TValue,TError> throws InvalidOperationException exception:

```
var failResult = Result.Fail<string,int>(0);
var value = failResult.Value; //throws exception
```

Accessing the Error property of an ok Result<TValue,TError> throws InvalidOperationException exception:

```
var okResult = Result.Ok<string,int>("some value");
var error = okResult.Error; //throws exception
```

You can also use the From method to create a Result<TValue,TError>. It accepts a predicate function, a value and an error. If the predicate evaluates to true the method returns an ok Result<TValue,TError> containing the value, if the predicate evaluates to false it returns a fail Result<TValue,TError> containing the error:

```
var okResult = Result.From<string,int>(()=>true, "some value",0);       // creates an␣
↪ok Result<string,int> with "some value" as the value
var failResult = Result.From<string,int>(()=>false, "some value",0);    // creates a␣
↪fail Result<string,int> with 0 as the error
```

CHAPTER 4

HttpResult monad

Please read *about the result monad* first because the HttpResult monad is essentially the same with an added state property of type *IHttpState*. As the result monad, there are four variations of the HttpResult monad:

- HttpResult
- HttpResultWithError<TError>
- HttpResult<TValue>
- HttpResult<TValue,TError>

All variations of the HttpResult monad contain an IsSuccess and it's inverse IsFailure property that indicate if the operation was successful or not; as well as an HttpState property that represents the http operation that was performed.

From a developers perspective there is one big difference between the HttpResult monad and the Result monad: whilst the Result monad can start to be used without any extra coding effort, the HttpResult monad requires a bit more coding effort. Please keep reading to understand how to take advantage of the HttpResult monad.

Installing

The HttpResult monad NuGet package can be found at HttpResultMonad NuGet. Installing is performed via NuGet:

```
PM> Install-Package HttpResultMonad
```

HttpResult monads: How To

Please read the *how-to for each variation of the Result monad* to understand how to use the HttpResult monads. The only difference is that the HttpResult also requires an HttpState in its Ok and Fail methods. See *the HttpState* to understand how to create an IHttpState instance.

To use this you should create a wrapper on the methods that do your http communication. For instance, if you are using the System.Net.Http.HttpClient class you could do the following:

```
private HttpClient _httpClient = new HttpClient();

public async Task<HttpResult> SendAsync(
        HttpRequestMessage request,
        CancellationToken cancellationToken = default(CancellationToken))
{
    var response = await _httpClient
                    .SendAsync(request, cancellationToken)
                    .ConfigureAwait(false);

    IHttpState httpState = new HttpClientState(response);
    return response.IsSuccessStatusCode
        ? HttpResult.Ok(httpState)
        : HttpResult.Fail(httpState);
}
```

The above example shows the idea for using the HttpResult class based on System.Net.Http.HttpClient. This code can be found at HttpResultClient class and you can use it by installing HttpResultMonad.HttpResultOnHttpClient NuGet.

Obviously you can use other implementations but what is important is that you are responsible for capturing the IHttpState as well as deciding whether you should return an ok or a fail HttpResult. This was just a very simple and generic example. Let's say that you knew that you only deal with json responses and you want to have type safety. Then you could implement something like:

```
public async Task<HttpResult<T>> SendAsync<T>(
        HttpRequestMessage request,
        CancellationToken cancellationToken = default(CancellationToken)) where T :␣
↪class
{
    var response = await _httpClient
                    .SendAsync(request, cancellationToken)
                    .ConfigureAwait(false);

    var jsonBody = await response.Content.ReadAsStringAsync();
    var deserializedObj = /*deserialize the string jsonBody into an instance of type␣
↪T*/

    IHttpState httpState = new HttpClientState(response);
    return response.IsSuccessStatusCode
        ? HttpResult.Ok<T>(deserializedObj,httpState)
        : HttpResult.Fail<T>(httpState);
}
```

It is up to you to chose the best integration of the HttpResult monad with the class of your choosing that does the http communication in order to meat your requirements.

IHttpState: How To

The IHttpState represents an http operation. See here for the specification of this interface.

As explained in the previous section, to use HttpResult you will have to wrap the methods that you use do the http communication with it. When doing that you will also decide whether or not to capture the http state. If you do not pass any http state into the OK and Fail methods of the HttpResult then the http state will always be an empty http state.

See the HttpClientState class for an implementation of the IHttpState that is used with an HttpResult that

uses the System.Net.Http.HttpClient. This HttpClientState implements the IHttpState interface based on System.Net.Http.HttpResponseMessage. Note that the Dispose method is responsible for disposing the System.Net.Http.HttpResponseMessage and System.Net.Http.HttpRequestMessage. Usually you do this by applying a using statement such as:

```
private HttpClient _httpClient = new HttpClient();

using(var httpResponse = await _httpClient.GetAsync("https://github.com"))
{
    //do something with the httpResponse
}
```

However in the HttpResultClient class the HttpResponse is not disposed because the implementation of the IHttpState interface by the HttpClientState class does not immediately retrieve everything it needs to from it, namely the body. This was an implementation choice. I could have decided that I was happy with loading upfront the whole request and response bodies into the HttpResultClient class. If I did that than I could dispose of the HttpResponseMessage on the HttpResultClient class and the implementation of the Dispose method in the HttpClientState could be empty.

It is up to you to chose the best implementation for IHttpState that meats your requirements.

If you need to create an empty state do:

```
using HttpResultMonad.State;

var emptyState = HttpState.Empty;
```

To check if an HttpState is empty do:

```
var emptyState = HttpState.Empty;
var isEmptyState = emptyState.Equals(HttpState.Empty); //evaluates to true
```

HttpResult monad is disposable

The Dispose method of the HttpResult just calls the Dispose method on its HttpState property which is of type IHttpState. This means two things:

- You should always dispose the HttpResult instances once you don't need them anymore.
- When implementing the IHttpState you should take care of releasing any disposable resources.

CHAPTER 5

Combine methods

The Combine methods are present for the Result and HttpResult monads. They allow you to evaluate a group of results and determine if they are all successful or not.

Combine methods for Result monads

the available combine methods for the Result monads are:

```
public static Result Combine(params Result[] results);

public static Result Combine<T>(params Result<T>[] results);

public static ResultWithError<TError> Combine<TError>(params ResultWithError<TError>
↪[] resultsWithError);

public static ResultWithError<TError> Combine<TValue, TError>(params Result<TValue,␣
↪TError>[] results);
```

The first method combines an array of Result instances and returns an ok Result if all results are ok, otherwise it returns a failed Result;

The second method combines an array of Result<T> instances and returns an ok Result if all results are ok, otherwise it returns a failed Result. Note that it does not return a Result<T>. That's because even if in the case where at least one Result<T> is a fail result it could return the first failure, for the scenario where all are ok Result<T> instances it does not seem right to randomly chose one of the ok results.

The third method combines an array of ResultWithError<TError> instances and returns an ok ResultWithError<TError> if all results are ok, otherwise it returns the first failed ResultWithError<TError> in the array. In contrast with the previous method it does not "reduce" the ResultWithError<TError> to a Result because an ok ResultWithError does not have a value for Error. In other words all ok instances of ResultWithError<TError> with the same type of TError are equal.

The fourth method combines an array of Result<TValue, TError> instances and returns an ok ResultWithError<TError> if all results are ok, otherwise it returns a failed ResultWithError<TError> where the error is from the

first fail Result<TValue, TError> in the array. Note that it does not return a Result<TValue, TError>. That's because even if in the case where at least one Result<T> is a fail result it could return the first failure, for the scenario where all are ok Result<TValue, TError>[] instances it does not seem right to randomly chose one of the ok results.

Combine methods for HttpResult monads

The available combine methods for the HttpResult monads are:

```
public static HttpResult Combine(params HttpResult[] results);

public static HttpResult Combine<T>(params HttpResult<T>[] results);

public static HttpResultWithError<TError> Combine<TError>(params HttpResultWithError
↪<TError>[] resultsWithError);

public static HttpResultWithError<TError> Combine<TValue, TError>(params HttpResult
↪<TValue, TError>[] results);
```

The first method combines an array of HttpResult instances and returns an ok HttpResult if all results are ok, otherwise it returns the first failed HttpResult. In the case where all HttpResult instances are ok the returned ok HttpResult instance has an empty HttpState.

The second method combines an array of HttpResult<T> instances and returns an ok HttpResult if all results are ok, otherwise it returns a failed HttpResult. Note that it does not return a HttpResult<T>. That's because even if in the case where at least one HttpResult<T> is a fail result it could return the first failure, for the scenario where all are ok HttpResult<T> instances it does not seem right to randomly chose one of the ok results. In the case where all HttpResult instances are ok the returned ok HttpResult instance has an empty HttpState. Otherwise the failed HttpResult will contain the HttpState of the first fail HttpResult<T> in the array.

The third method combines an array of ResultWithError<TError> instances and returns an ok ResultWithError<TError> if all results are ok, otherwise it returns the first failed ResultWithError<TError> in the array. In contrast with the previous method it does not "reduce" the ResultWithError<TError> to a Result because an ok ResultWithError does not have a value for Error. In other words all ok instances of ResultWithError<TError> with the same type of TError are equal. In the case where all HttpResult instances are ok the returned ok HttpResult instance has an empty HttpState.

The fourth method combines an array of HttpResult<TValue, TError> instances and returns an ok HttpResultWithError<TError> if all results are ok, otherwise it returns a failed HttpResultWithError<TError> where the error is from the first fail HttpResult<TValue, TError> in the array. Note that it does not return a HttpResult<TValue, TError>. That's because even if in the case where at least one HttpResult<T> is a fail result it could return the first failure, for the scenario where all are ok HttpResult<TValue, TError>[] instances it does not seem right to randomly chose one of the ok results. In the case where all HttpResult instances are ok the returned ok HttpResult instance has an empty HttpState.

Examples

Here are a some examples of using the Combine methods on the Result monads. The usage of Combine methods for HttpResult monads is similar.

Combining a set of Result instances:

```
var resultsLists = new List<Result>
{
    Result.Ok(),
```

```
    Result.Fail(),
    Result.Ok()
};

Result combinedResult = Result.Combine(resultsLists.ToArray()); //at least one Result␣
↪is fail so the combinedResult is a fail Result
```

Combining a set of Result<TValue> instances:

```
var resultsLists = new List<Result<string>>
{
    Result.Ok("value1"),
    Result.Ok("value2"),
    Result.Ok("value3")
};

Result combinedResult = Result.Combine(resultsLists.ToArray());
/*
*  all Result<string> are ok so the combinedResult is an ok Result
*  it is not a Result<string> because it doesn't seem right to randomly chose one of␣
↪the values.
*/
```

Combining a set of Result<TValue,TError> instances:

```
var resultsLists = new List<Result<int, string>>
    {
        Result.Ok<int,string>(1),
        Result.Fail<int, string>("first error");
        Result.Ok<int,string>(2),
        Result.Fail<int,string>("second error")
    };

    ResultWithError<string> combinedResult = Result.Combine(resultsLists.ToArray());
    /*
    *  at least one Result<int, string> is fail so the combinedResult is a fail␣
↪ResultWithError<string>
    *  combinedResult.Error evaluates to "first error"
    */
```

CHAPTER 6

Extensions

The extension methods available don't cover every possible scenario. I encourage you to take a look at the code in the csharp-functional repository and create your own.

One thing to note is that the extensions can perform transformations on the types being extended. What I mean is that an extension can be applied to a Result<T> and return a Result<K> or be applied to a Maybe<T> and return a Result<T>.

I think of these transformations in two kinds:

- Generic type transformation: for instance from Result<T> to Result<K>. T became K.
- Monad transformation: for instance from Maybe<T> to Result<T>

For generic type transformations you will need to provide a function for each type you want to transform. For instance an extension like:

```
public static Result<KValue> OnSuccessToResultWithValue<TValue, KValue>(
    this Result<TValue> result,
    Func<TValue, KValue> onSuccessFunc)
{
    if (result.IsFailure)
    {
        return Result.Fail<KValue>();
    }

    var newValue = onSuccessFunc(result.Value);
    return Result.Ok<KValue>(newValue);
}
```

Mutates KValue to TValue by applying the onSuccessFunc to the instance of type TValue. For monad transformations you might not need any extra function if it's implicitly known how to create one monad from the other:

```
public static Result<T> ToResultWithValue<T>(this Maybe<T> maybe)
{
        /*the transformation from Maybe<T> to Result<T> is implicit, no function is␣
↪required*/
```

```
        return maybe.HasNoValue
        ? Result.Fail<T>()
        : Result.Ok(maybe.Value);
}
```

Since I know how to create a Result<T> from a Maybe<T> then no function is required. However if this transformation was not implicit then a function would have to be passed in to be applied and transform from one monad to another.

In the NuGet packages there are extensions that mutate from Maybe monad to Result, Result<T>, Result<TValue,TError>; as well as many others that mutate from a variation of Result monad to a variation of HttpResult monad like from Result<TValue,TError> to HttpResult<KValue,Terror>.

Note: The underlying statement that derives from this introduction is that as a rule of thumb you should perform transformations that require only one function because of code readability:

- If you're doing a generic type transformation and at the same time a monad transformation then the monad transformation should be implicit so that you will only require the function to mutate the generic type.
- There is no problem in breaking this advice. You can even mutate more than one generic type at a time and go from Monad<A,B,C> to Monad<X,Y,Z> but at the expense of having 3 mutating functions: A->X, B->Y and C->Z.
- The more functions that need to be passed in to the extension method the worst the readability of the code becomes. This is subject to personal opinion but I think that due to the C# syntax it becomes harder to read the code if you start to have many functions passed in as arguments on chained method calls.

Note: Consider for a moment the OnSuccess extension methods. There can be many overloads for OnSuccess which can only be distinguished by the returning type and in C# we can not have two methods with equal signature (the return type is not part of a method's signature). To overcome this, what I've chosen to do is to distinguish them by changing the method name to cater for different return types. Therefore you find different OnSuccess methods with appended ToX where X relates to the return type, such as OnSuccessToResultWithValue. This applies to all extension methods.

NuGet packages

The extension methods are divided into the following NuGet packages:

- ResultMonad.Extensions NuGet : extensions to the result monads that always return another instance of a result monad.
- HttpResultMonad.Extensions NuGet : extensions to the http result monads that always return another instance of an http result monad.
- ResultMonad.Extensions.HttpResultMonad NuGet : extensions to the result monad that return an instance of an http result monad.
- MaybeMonad.Extensions.ResultMonad NuGet : extensions to the maybe monad that return an instance of a result monad.

The ResultMonad.Extensions.HttpResultMonad NuGet is separated from the ResultMonad.Extensions NuGet, just as the ResultMonad.Extensions.HttpResultMonad NuGet is separated from the HttpResultMonad.Extensions NuGet, because it allows you to only work with either the Result monad or the HttpResult monad. If they were together then installing extensions for the result monad would mean that you would have to install the http result monad NuGet as well even though you didn't want to use it.

Installing

Installing is performed via NuGet. For example to install the ResultMonad.Extensions NuGet do:

```
PM> Install-Package ResultMonad.Extensions
```

Maybe extensions: examples

The Maybe extensions allow you to map from a maybe monad to a result monad.

Example usages are:

```
Maybe<User> maybeUser = GetUserById(id);
Result result1 = maybeUser.ToResult();
/*
* if maybeUser.HasValue is true then result1.IsSuccess is true.
* if maybeUser.HasValue is false then result1.IsFailure is true.
*/

Result<User> result2 = maybeUser.ToResultWithValue();
/*
* if maybeUser.HasValue is true then result2.IsSuccess is true and result2.Value␣
↪evaluates to maybeUser.Value.
* if maybeUser.HasValue is false then result2.IsFailure is true.
*/

Result<User,string> result3 = maybeUser.ToResultWithValueAndError(()=>"no user found
↪"); //result3.Value evaluates to maybeUser.Value if ;
/*
* if maybeUser.HasValue is true then result3.IsSuccess is true and result3.Value␣
↪evaluates to maybeUser.Value.
* if maybeUser.HasValue is false then result3.IsFailure is true and result3.Value␣
↪evaluates to "no user found".
*/
```

Result and HttpResult extensions: examples

Note: keep in mind that:

- The examples are focused on the Result monad but the same applies for the HttpResult monad.
- Although the examples shown are for the type Result<TValue,TError> the same concepts apply to the other variations of the Result monad.

The available extensions are of the type:

- OnSuccess: executes if the IsSuccess property of the result is true;
- OnError: executes if the IsFailure property of the result is true;
- Ensure: evaluates a condition and executes if the condition is true;

One that is not yet implemented but is definitely useful would be the OnBoth. OnBoth would execute regardless of the IsSuccess/IsFailure of the result.

OnSuccess example

An example of using an OnSuccess extension is:

```
public static Result<PlaylistId, PlaylistIdError> Create(int playlistId)
{
        return Result.From(() => playlistId >= 0, playlistId, PlaylistIdError.
↪CanNotBeNegative)
                .OnSuccessToResultWithValueAndError(id => new PlaylistId(id));
}
```

In this example we start by creating a result with the From method. The From method will check if the variable playlistId is positive and if it is then creates an ok Result<int,PlaylistIdError> with its property Value evaluating to the value of playlistId; if playlistId is negative it creates a fail Result<int,PlaylistIdError> with the Error property evaluating to PlaylistIdError.CanNotBeNegative.

After the call to OnSuccessToResultWithValueAndError will only execute if the result is a an ok result. If it is an ok result the onSuccess function is executed wand creates an ok Result<PlaylistId,PlaylistIdError> with an instance of PlaylistId in its Value property; if it is a fail result the onSuccess function is not executed and a fail Result<PlaylistId,PlaylistIdError> is created and the error from the previous result is propagated, which means the property Error evaluates to PlaylistIdError.CanNotBeNegative.

Ensure example

An example of using an Ensure extension is:

```
public static Result<SearchQuery, SearchQueryError> Create(string searchQuery)
{
        return Result.From(() => !string.IsNullOrEmpty(searchQuery), searchQuery,␣
↪SearchQueryError.CanNotBeNullOrEmpty)
                .EnsureToResultWithValueAndError(query => query.Length <= 100,␣
↪SearchQueryError.CanNotBeMoreThan100CharactersLong)
                .OnSuccessToResultWithValueAndError(query => new SearchQuery(query));
}
```

In this example we start by creating a result with the From method. The From method will check if the variable searchQuery has a value and if it has then creates an ok Result<string,SearchQueryError> with its property Value evaluating to the value of searchQuery; if searchQuery is null or empty it creates a fail Result<string,SearchQueryError> with the Error property evaluating to SearchQueryError.CanNotBeNullOrEmpty.

After the call to EnsureToResultWithValueAndError will only execute if the result is a an ok result. If it is an ok result the predicate function is executed and if it evaluates to true then an ok Result<string,SearchQueryError> probating the value from the previous result with an instance; if it is an ok result but the predicate function evaluates to false then it creates a fail Result<string,SearchQueryError> and the Error property set to SearchQueryError.CanNotBeMoreThan100CharactersLong; if the previous result is a fail result then predicate function is not executed and a fail Result<string,SearchQueryError> is created and the Error property set to SearchQueryError.CanNotBeMoreThan100CharactersLong.

In the end the OnSuccessToResultWithValueAndError is executed which follows the rules explained in the previous example.

OnError example

An example of using an OnError extension is:

```
/*reusing here the SearchQuery.Create method from the previous example*/
var tag = "a tag";
var query = "a query";

Result<PlaylistSearchRequest,PlaylistSearchRequestError> result = SearchQuery
                                                                    .
 ↪Create(searchQuery)
                                                                    .
 ↪OnErrorToResultWithValueAndError(PlaylistSearchRequestError.From)
                                                                    .
 ↪OnSuccessToResultWithValueAndError(query => new PlaylistSearchRequest(query, tag));

/*here is the declaration for the method used on the onError function:
*
* public static PlaylistSearchRequestError From(SearchQueryError error)
*
*/
```

In this example we start by creating a Result<SearchQuery,SearchQueryError> with the method SearchQuery.Create.

After the call to OnErrorToResultWithValueAndError will only execute if the result is a a fail result. If it is a fail result the onError function is executed and returns a Result<SearchQuery,PlaylistSearchRequestError> by converting the SearchQueryError instance to a PlaylistSearchRequestError; if the result returned from the SearchQuery.Create method is an ok result then the onError function is not executed and an ok Result<SearchQuery,PlaylistSearchRequestError> is returned. In this case the Value property of the new ok result contains the value from the previous result.

In the end the OnSuccessToResultWithValueAndError is executed which follows the rules explained in the previous example.

Note: Notice that the TValue and TError types do not have constrains, they can be whatever you want. In these examples the TValue were primitive types (int, string) as well as custom classes (PlaylistId, SearchQuery, PlaylistSearchRequest). In these examples the TError were only custom classes (PlaylistIdError, SearchQueryError, PlaylistSearchRequestError) but they could be whatever you would like to use like Enums for instance.

Mapping from one monad type to another

In your application you can use different types of monads at the same type. You can use all the ones mentioned in this documentation and even others you have defined. To make them work together you will need a way to map from one type to another.

The extensions available will tell you the type that will be returned. Some extensions will not perform any mapping and some will. For instance you can have one extension that takes a Result<TValue> and returns a ResultWithError<TError>. These transformations are indicated by the ToX appended to the extension method names.

Another example is going from a Maybe type to a Result type. You can call ToResult() on a Maybe instance and it will return you a Result instance. See *here* for more details.

These transformations are essential to allow the railway oriented programming style described in the next section.

Putting it all together: Railway Oriented Programming

Scott Wlaschin gave a great talk on NDC about this concept.

The description of the talk is:

"When you build real world applications, you are not always on the "happy path". You must deal with validation, logging, network and service errors, and other annoyances. How do you manage all this within a functional paradigm, when you can't use exceptions, or do early returns, and when you have no stateful data? This talk will demonstrate a common approach to this challenge, using a fun and easy-to-understand "railway oriented programming" analogy. You'll come away with insight into a powerful technique that handles errors in an elegant way using a simple, self-documenting design."

I advise you to watch the talk because it is very good and whatever I do to try and summarize the idea will never be as good.

For the purposes of this documentation I will give you two examples of how you can use the monad NuGets and its extensions. I will show the same method implemented with and without the use of the extensions and you will see that the code using extensions is much more compact and once you get used to this style of programming it is also a more expressive way of coding.

Example A1: in this example we want to get the reposters of a track. Let's do it without extension methods:

```
public async Task<HttpResult<UsersSet, GetTrackRepostersError>>␣
↪GetTrackRepostersAsync(
    int trackId,
    int limit,
    CancellationToken cancellationToken = default(CancellationToken))
{
    var trackIdResult = TrackId.Create(trackId);
    if (trackIdResult.IsFailure)
    {
        var trackIdError = GetTrackRepostersError.From(trackIdResult.Error);
        return HttpResult.Fail<UsersSet, GetTrackRepostersError>(trackIdError);
    }

    var limitResult = Limit.Create(limit);
    if (limitResult.IsFailure)
    {
        var limitError = GetTrackRepostersError.From(limitResult.Error);
        return HttpResult.Fail<UsersSet, GetTrackRepostersError>(limitError);
    }

    var requestUrl = _trackUrlBuilder.GetTrackRepostersUrl(trackIdResult.Value,␣
↪limitResult.Value);
    var httpGetResult = await HttpClient.GetAsync<UsersSet>(requestUrl,␣
↪cancellationToken);
    if (httpGetResult.IsFailure)
    {
        return HttpResult.Fail<UsersSet, GetTrackRepostersError>
↪(GetTrackRepostersError.HttpFail);
    }

    var usersSet = httpGetResult.Value;
    return HttpResult.Ok<UsersSet, GetTrackRepostersError>(usersSet);
}
```

Example A2: now with extension methods:

```
public Task<HttpResult<UsersSet, GetTrackRepostersError>> GetTrackRepostersAsync(
    int trackId,
    int limit,
    CancellationToken cancellationToken = default(CancellationToken))
```

```
{
    var trackIdResult = TrackId.Create(trackId);
    var limitResult = Limit.Create(limit);
    var inputResult = Result.Combine(
        trackIdResult.OnErrorToResultWithError(GetTrackRepostersError.From),
        limitResult.OnErrorToResultWithError(GetTrackRepostersError.From));

    return inputResult
        .OnSuccessToHttpResultWithValueAndError(() =>
        {
            var requestUrl = _trackUrlFactory.GetTrackRepostersUrl(trackIdResult.
↪Value, limitResult.Value);
            return HttpClient
                .GetAsync<UsersSet>(requestUrl, cancellationToken)
                .OnErrorToHttpResultWithValueAndError(() => GetTrackRepostersError.
↪HttpFail);
        });
}
```

Here are the methods used in the examples A1 and A2:

```
public static Result<TrackId, TrackIdError> Create(int trackId);

public static Result<Limit, LimitError> Create(int limit);

public static GetTrackRepostersError From(TrackIdError trackIdError);

public static GetTrackRepostersError From(LimitError limitError);

public string GetTrackRepostersUrl(TrackId trackId, Limit limit);

public Task<HttpResult<T>> GetAsync<T>(string requestUrl, CancellationToken␣
↪cancellationToken) where T : class

public static GetTrackRepostersError HttpFail { get; }
```

I'll explain the code in both examples but focusing on the example with extension methods.

The first thing is to validate the input arguments and we do that by creating an inputResult that combines the results from the TrackId.Create and Limit.Create methods. If both results are ok then the inputResult is an ok ResultWithError<GetTrackRepostersError>; if at least one is a fail result then the inputResult is a fail ResultWithError<GetTrackRepostersError> and its Error property is set to the first failure result of the array passed into the Result.Combine method.

Now we can call OnSuccessToHttpResultWithValueAndError which is only executed if the inputResult is an ok result. If it is a fail result then the onSuccess function is not executed and it returns a fail Task<HttpResult<UsersSet, GetTrackRepostersError>> with the Error property set to the error that was on the inputResult. If the inputResult is an ok result then the onSuccess function is executed and what it is doing is creating an url to do the http call, calling HttpClient.GetAsync which will return a Task<HttpResult<UsersSet>>.

Finally if the result from the call to HttpClient.GetAsync is an ok result then OnErrorToHttpResultWithValueAndError is not executed and an ok Task<HttpResult<UsersSet, GetTrackRepostersError>> is returned. Its Value property is set to the value that was on the previous result returned from the call to HttpClient.GetAsync. If an fail result was returned from the call to HttpClient.GetAsync then the onError function is executed and a fail Task<HttpResult<UsersSet, GetTrackRepostersError>> is returned with its Error property set to GetTrackRepostersError.HttpFail.

Example B1: in this example we want to do a sign up. Let's do it without extension methods:

```
public async Task<HttpResult<SignUp, SignUpError>> SignUpAsync(
        SignUpRequest signUpRequest,
        CancellationToken cancellationToken = default(CancellationToken))
{
    var signUpRequestResult = Result.From(() => signUpRequest != null, signUpRequest,␣
↪SignUpError.SignUpRequestCanNotBeNull);
    if (signUpRequestResult.IsFailure)
    {
        return HttpResult.Fail<SignUp, SignUpError>(signUpRequestResult.Error);
    }

    var emailSignUpTokenResult = await GetEmailSignUpTokenAsync(signUpRequest.Email,␣
↪cancellationToken);
    if (emailSignUpTokenResult.IsFailure)
    {
        return HttpResult.Fail<SignUp, SignUpError>(SignUpError.HttpFail);
    }

    var emailSignUpTokenResp = emailSignUpTokenResult.Value;
    var signUpResult = await SignUpAsync(emailSignUpTokenResp.EmailSignUpToken,␣
↪signUpRequest, cancellationToken);
    if (signUpResult.IsFailure)
    {
        return HttpResult.Fail<SignUp, SignUpError>(SignUpError.HttpFail);
    }

    var signUp = signUpResult.Value;
    return HttpResult.Ok<SignUp, SignUpError>(signUp);
}
```

Example B2: now with extension methods:

```
public Task<HttpResult<SignUp, SignUpError>> SignUpAsync(
    SignUpRequest signUpRequest,
    CancellationToken cancellationToken = default(CancellationToken))
{
    return Result.From(() => signUpRequest != null, signUpRequest, SignUpError.
↪SignUpRequestCanNotBeNull)
        .OnSuccessToHttpResultWithValueAndError(request =>
        {
            return GetEmailSignUpTokenAsync(signUpRequest.Email, cancellationToken)
                .OnErrorToHttpResultWithValueAndError(() => SignUpError.HttpFail);
        })
        .OnSuccessToHttpResultWithValueAndError(emailSignUpTokenResp =>
        {
            return SignUpAsync(emailSignUpTokenResp.EmailSignUpToken, signUpRequest,␣
↪cancellationToken)
                .OnErrorToHttpResultWithValueAndError(() => SignUpError.HttpFail);
        });
}
```

Here are the methods used in the examples B1 and B2:

```
private Task<HttpResult<EmailSignUpToken>> GetEmailSignUpTokenAsync(Email email,␣
↪CancellationToken cancellationToken)

private Task<HttpResult<SignUp>> SignUpAsync(string emailSignUpToken, SignUpRequest␣
↪signUpRequest, CancellationToken cancellationToken);
```

```
public static SignUpError SignUpRequestCanNotBeNull { get;}

public static SignUpError HttpFail { get;}
```

Again, I'll explain the code in both examples but focusing on the example with extension methods.

The first thing is to validate the input. If the signUpRequest is null then the Result.From will return a fail Result<SignUpRequest,SignUpError> with its Error property set to SignUpError.SignUpRequestCanNotBeNull. If the signupRequest is not null then the Result.From returns an ok Result<SignUpRequest,SignUpError> with its Value property set to what is passed in by signUpRequest.

After we call OnSuccessToHttpResultWithValueAndError. If the previous result was a fail result then the onSuccessFunction is not executed and it returns a fail Task<HttpResult<SignUp, SignUpError>> with the Error property propagated from what was in the previous result. If the previous result was an ok result then the onSuccessFunction is executed and it tries to get an email signup token. The call to GetEmailSignUpTokenAsync returns an Task<HttpResult<EmailSignUpToken>> which after the call to OnErrorToHttpResultWithValueAndError will either be an ok or a fail Task<HttpResult<EmailSignUpToken, SignUpError>>. If it's an ok result then the Value will contain the value from the result that was returned from the call to GetEmailSignUpTokenAsync; if it's a fail result then the Error property will contain the error from the onErrorFunction which is SignUpError.HttpFail;

Finally, another call to OnSuccessToHttpResultWithValueAndError. Again, if the previous result was a fail result then the onSuccessFunction is not executed and it returns a fail Task<HttpResult<SignUp, SignUpError>> with the Error property propagated from what was in the previous result. If the previous result was an ok result then the onSuccessFunction is executed and it tries to perform the signup. The call to SignUpAsync returns an Task<HttpResult<SignUp>> which after the call to OnErrorToHttpResultWithValueAndError will either be an ok or a fail Task<HttpResult<SignUp, SignUpError>>. If it's an ok result then the Value will contain the value from the result that was returned from the call to SignUpAsync; if it's a fail result then the Error property will contain the error from the onErrorFunction which is SignUpError.HttpFail;

CHAPTER 7

More resources

- Pluralsight course by Vladimir Khorikov: Applying Functional Principles in C# <https://app.pluralsight.com/library/courses/csharp-applying-functional-principles/table-of-contents>‘_ .
- Vladimir Khorikov’s repository related with the Pluralsight course mentioned above: https://github.com/vkhorikov/CSharpFunctionalExtensions.
- Eric Lippert’s series of articles on monads: https://ericlippert.com/category/monads. It’s composed of thirteen articles and each one helped a lot to understand monads a bit better.
- Wes Dyer article on monads: https://blogs.msdn.microsoft.com/wesdyer/2008/01/10/the-marvels-of-monads. This article is mentioned on *Monads, part two <https://ericlippert.com/2013/02/25/monads-part-two/>*, the second article in the series and it contains one of the most succinct and easier to grasp definitions about a monad:
  - “Another way to look at these generic types (monads) is that they are “amplifiers”. An “amplifier” is something that increases the representational power of their “underlying” type.”
- Another good and not mathematical definition of monad can be found at: http://www.introtorx.com/content/v1.0.10621.0/10_LeavingTheMonad.html. It says:
  - “For us, a monad is effectively a programming structure that represents computations. Generally a monadic structure allows you to chain together operators to produce a pipeline, just as we do with our extension methods. Monads are a kind of abstract data type constructor that encapsulate program logic instead of data in the domain model.”
- Scott Wlaschin’s article on Railway Oriented Programming: https://fsharpforfunandprofit.com/rop. As well as the links at the bottom of the article about monads in general.