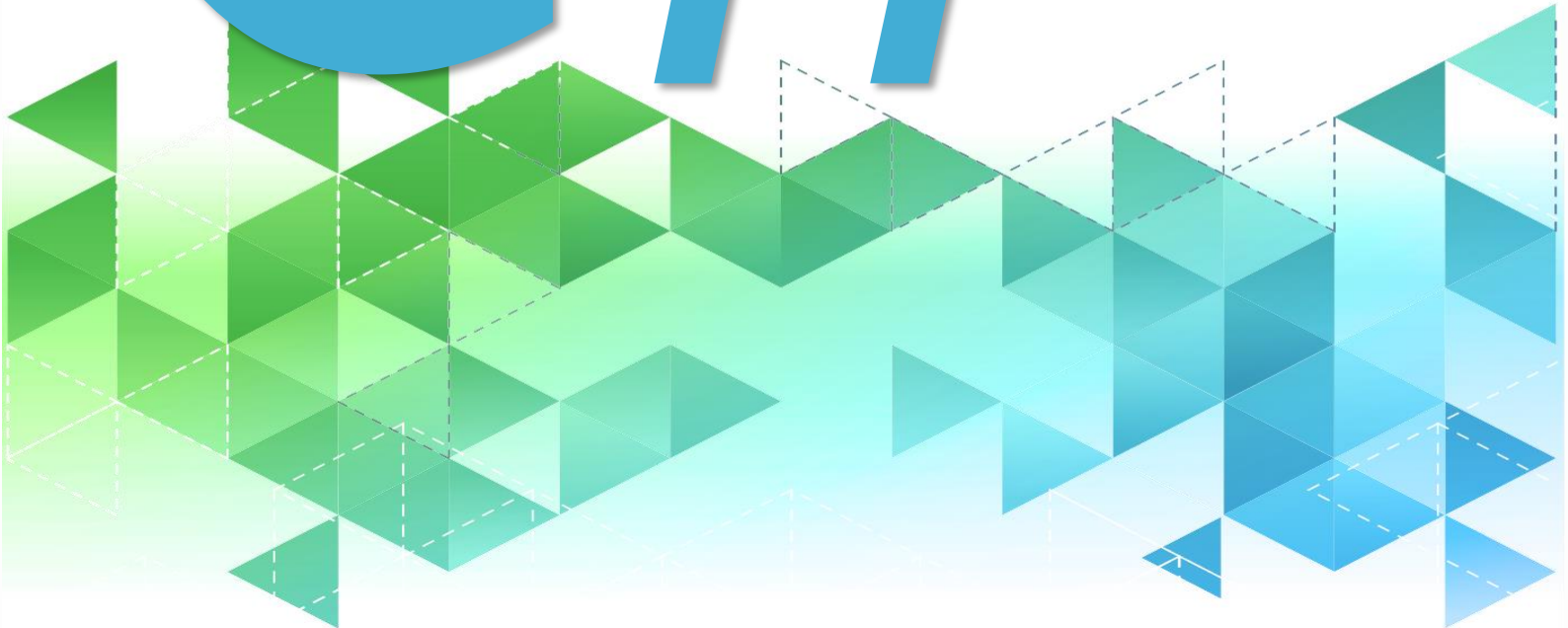


# Practical Functional Programming In C#

{with: language-ext}

A practical guide to the language-ext C# functional Library

# C#



# PRACTICAL FUNCTIONAL PROGRAMMING IN C#

A practical guide to the language-ext C# functional Library

Dimitris Papadimitriou

This version was published on 28-Apr-2021

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Feel free to contact me at:

<https://www.linkedin.com/in/dimitrispapadimitriou/>

<https://leanpub.com/u/dimitrispapadim>

<https://medium.com/@dimpapadim3>

<https://github.com/dimitris-papadimitriou-chr>

[dimitrispapadim@live.com](mailto:dimitrispapadim@live.com)

## Acknowledgments

<https://pixabay.com/illustrations/tiger-animal-stare-teeth-wild-2823181/>

© 2021 Dimitris Papadimitriou

# Contents

The <i>language-ext</i> C# library .....	3
1 Language Functional support.....	4
1.1 Built-in .NET Delegates.....	4
1.2 Higher-order functions .....	7
2 Functors.....	9
2.1 The Identity Functor .....	10
2.2 Commutative Diagrams .....	12
2.3 Maybe Functor aka Option<T>.....	13
2.3.1 Maybe Functor Example .....	13
2.3.2 Maybe in Language-ext / Option .....	15
2.3.3 Maybe Functor Example With language-ext Option .....	15
2.3.4 C# 8. pattern matching Support for Option.....	16
2.3.5 Folding Maybe .....	16
2.3.6 Using the Linq syntax.....	17
3 Monads.....	18
3.1 Maybe Monad .....	20
3.1.1 Using language-ext Option.....	22
4 A Clean Functional Architecture Example .....	24
4.1 Download and Setup the Project.....	24
4.1.1 Clean Architecture with .NET core .....	25
4.2 A Functional Applications Architecture .....	27
4.3 The Contoso Clean Architecture with .NET core and language-ext.....	28
4.4 Web Api .....	29

# Purpose

OO makes code understandable by encapsulating moving parts.  
FP makes code understandable by minimizing moving parts.  
—Michael Feathers (Twitter)

This book is aiming to present the basics of functional programming using the **language-ext** library. We will try to exhibit the usage of the basic Functional types: Option, Either, Task and Validation within an ASP.NET MVC web paradigm.

There is a **WebApplicationExample** solution that has multiple simpler examples of language-ext in a .net core web application. The intention is to build up an understanding behind the architecture of the **Sample Contoso** web Application in the **language-ext GitHub project** which is examined in the last chapter of this book.

## The *language-ext* C# library

This book is all about the **specialization** of the concepts of functional programming with the language-ext C# library. There are some functional libraries out there for C#, but currently the only complete library that could be used in a commercial project is the language-ext. In this section we are going to see a couple of the basic crosscutting ideas in the library and then in the following Chapters we will introduce all the different structures.

In this book we are going only to **use the Nu-get package LanguageExt.Core**.

# C# Functional Features

## An overview

"1996 - James Gosling invents Java. Java is a relatively verbose, garbage collected, class based, statically typed, single dispatch, object oriented language with single implementation inheritance and multiple interface inheritance. Sun loudly heralds Java's novelty.

2001 - Anders Hejlsberg invents C#. C# is a relatively verbose, garbage collected, class based, statically typed, single dispatch, object oriented language with single implementation inheritance and multiple interface inheritance. Microsoft loudly heralds C#'s novelty."

*-Brief, Incomplete and Mostly Wrong History of Programming Languages*

## 1 Language Functional support

### 1.1 Built-in .NET Delegates

.NET contains a set of delegates designed for commonly occurring use cases, so we do not have to declare custom delegates. Those are residing in the System namespace. Here are the most important and most used :

---

#### Func<T,R>

The most common delegate by far will be the `Func<T, R>`, which represents functions and methods that look like this `R f(T t)` taking one argument and returning one result.

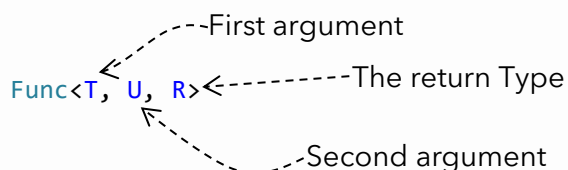
Visually this could be represented by just an arrow:  $T \xrightarrow{f} R$

---

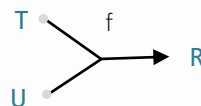
#### Func<T,U,R>

Represents a function Type that has the following signature `R f(T t1, U t2)`.

The `Func<T,U,R>` represents a Delegate Type that has the following signature  $(T, U) \rightarrow R$



If you wanted to visualize this could be a join:



## Action<T>

The `Action<T>` Functional interface represents functions that have this signature `void f(T t)`. In functional programming we usually do not like the `void` because it is not a Type like all the others, and this forces us to treat it differently. `Action<T>` is just a `Func<T, void>` but because `void` is not a Type .NET had to invent `Action<T>`. As an example of usage look at the following, where we "consume" a string returning nothing:

```
Action<string> print = (string x) => Console.WriteLine(x);
```

**When you return `void` is an indication that you create a side effect of some sort.** We can shorten the declaration, using the point free notation :

```
Action<string> print1 = Console.Write;
```

A visualization for `Action<T>` could be:

The diagram shows a point `T` on the left. An arrow labeled `f` points from `T` to a solid black circle. A dashed arrow points from the text "This represents the `void`" to the solid black circle.

## Func<T>

The `Func<T>` represents functions that have this signature `T f()` that do not take any arguments. A simple example would be:

```
Func<string> getName = () => "jim";
```

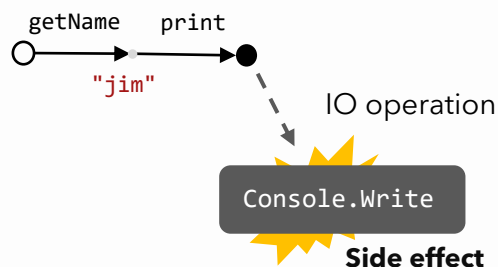
visualize this as a value coming out of nothing:

The diagram shows a circle on the left. An arrow labeled `f` points from the circle to the output `T`.

`Func<T>` is the dual of `Action<T>` and in this way we can compose them and annihilate them into a `void`.

```
Action<string> print = Console.Write;
Func<string> getName = ()=>"jim";

print(getName());           // prints Jim as side effect with return type void
```




---

## Predicate<T>

The `Predicate<T>` Functional interface represents functions that have this signature `bool f(T t1)` that returns a `bool`. This is equivalent to a `Func<T, bool>`.

---

## Summary Table

<code>Action&lt;T1&gt;</code>	<code>void f(T1 a){...}</code>
<code>Action&lt;T1,T2&gt;</code>	<code>void f(T1 arg1, T2 arg2){...}</code>
<code>Func&lt;TResult&gt;</code>	<code>TResult f(){...}</code>
<code>Func&lt;T1,TResult&gt;</code>	<code>TResult f(T1 arg1){...}</code>
<code>Func&lt;T1,T2,TResult&gt;</code>	<code>TResult f(T1 arg1, T2 arg2){...}</code>
<code>Func&lt;T1,T2,...T16,TResult&gt;</code>	<code>TResult f(T1 arg1, T2 arg2,...,T16 arg16){...}</code>
<code>Predicate&lt;T&gt;</code>	<code>bool f(T arg1){...}</code>


## Remember :

1. The `Action` delegates only have input arguments.
2. The `Func<_,_,_>` is the general case.
3. For Any `Func<_,_,TResult>` the Result is always at the Last position.
4. If you only want to return a result `()=>...` you must use a `Func<TResult>`.

## 1.2 Higher-order functions


A **Higher-Order function (Hof)** is a function that receives a function as an argument or returns a function as output. But I assume that you are already familiar with this definition.

The following is a Hof because it takes a `Func<T, U>` delegate as an argument.



```
R F<T, U, R>(Func<T, U> g, T y)
{
    ...
}
```

The type of this function is  $\mathbf{F}: ((\mathbf{T} \rightarrow \mathbf{U}) \times \mathbf{T}) \rightarrow \mathbf{R}$ . While the following is a Hof because it returns a `Func<T, U>`



```
Func<T, U> G<T, U>(T y)
{
    ...
}
```

With a type of  $\mathbf{G}: \mathbf{T} \rightarrow (\mathbf{T} \rightarrow \mathbf{U})$ , we can also use the local function form inside other methods or functions.

In the form of `Func` delegates the previous functions now should be declared like this:

```
Func<Func<T, U>, T, R> F = (Func<T, U> g, T y) => ...
```

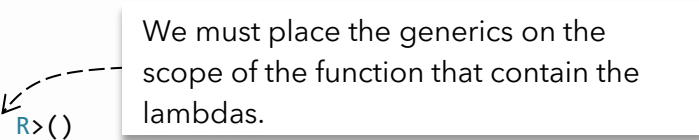
```
Func<T, Func<T, U>> G = (T y) => ...
```

unfortunately, C# does not allow for Generic definitions at the level of the delegate. We cannot write for example :

```
Func<Func<T, U>, T, R> F = <T, U, R> (Func<T, U> g, T y) => ...
```

```
Func<T, Func<T, U>> G = <T, U> (T y) => ...
```

For this reason, we must place them in a Function or Class, and include the generic constraints there:



```
public static void Foo<T, U, R>()
{
    Func<Func<T, U>, T, R> F = (Func<T, U> g, T y) => ...
}
```

```

    Func<T, Func<T, U>> G = (T y) => ...
}

```

Again, here the type of `Func<Func<T, U>, T, R>` corresponds to  $((\mathbf{T} \rightarrow \mathbf{U}) \times \mathbf{T}) \rightarrow \mathbf{R}$  and the type of `Func<T, Func<T, U>>` corresponds to  $\mathbf{T} \rightarrow (\mathbf{T} \rightarrow \mathbf{U})$

Obviously, we can have as many levels of Higher order Functions by allowing the Functions passed as arguments or return as results being themselves of Higher type. For example:

```

R F<T, U, R>(Func<Func<T, U>, U> g, T y)
{
    ...
}

```

This is a Hof passed as an argument.

With type **F**:  $((\mathbf{T} \rightarrow \mathbf{U}) \rightarrow \mathbf{U}) \times \mathbf{T} \rightarrow \mathbf{R}$ . In practice this type of Higher-Higher-order-functions becomes too convoluted too fast, and it is not recommended. In a delegate variable declaration this already looks way to complex.

```

Func<Func<Func<T, U>, U>, T, R> F = (Func<Func<T, U>, U> g, T y) => ...

```

What makes it worse is that we cannot replace the variable type with the `var` keyword.

# Functors

---

“It should be observed first that the whole concept of a category is essentially an auxiliary one; Our basic concepts are essentially those of a functor and of a natural transformation.”  
S. Eilenberg and S. MacLane

## The Idea:

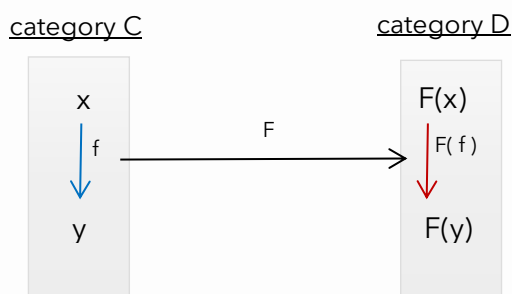
In C# the most famous functional programming idea is to use `List.Select` to replace iterations instead of *for loops* in order to transform the values of the array. That is because an array is a Functor, which is a more abstract idea that we will explore in this section.

“Practically a Functor is anything that has a valid **.Map(f)** method.”

Functors can be considered the core concept of category theory.

## 2 Functors

In mathematics, a **functor** is a map between categories.



This Functor  $F$  must map two requirements.

1. Map each object  $x$  in  $C$  with an object  $F(x)$  in  $D$ ,
2. Map each morphism  $f$  in  $C$  with a morphism  $F(f)$  in  $D$

For object-oriented programming, the best metaphor for functors is a **container**, together with a **mapping** function. The Array as a **data structure** is a Functor, **together** with the **map** method. The **map** is the array method that transforms the items of the array by applying a function  $f$ .

## 2.1 The Identity Functor

We will start by looking at the minimum structure that qualifies as a functor in C#:

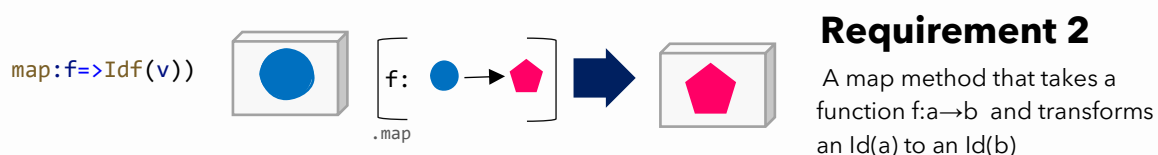
```
public class Id<T>
{
    public T Value { get; set; }
    public Id(T value) => Value = value;
    public Id<T1> Map<T1>(Func<T, T1> f) => new Id<T1>(f(Value));
}
```

that's requirement 1

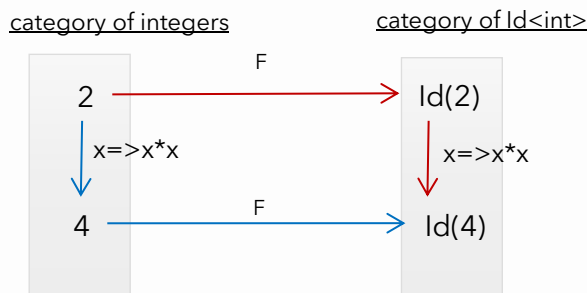
that's requirement 2

This is the minimum construction that we could call a functor because it has exactly two things:

1. A **"constructor"** that maps a value  $v$  to an object literal `Id<T>`
2. and it has a **mapping** method `Map<T1>(Func<T, T1> f)` that lifts functions  $f$



Because it's the minimal functor structure it goes by the name **Identity functor**. Let us see a simple example where we have two integers 2 and 4 (here we take for simplicity the category of integers as our initial category  $C$ ) also in this category there is the function `square = x => x*x` that maps 2 to 4.



If we apply the `Id(_)` constructor we can map each integer to the `Id<int>` category. For example, 2 will be mapped to `Id(2)` and 4 maps to `Id(4)`, the only part missing is the correct lifting of the function `f` `Id(f)` to this new category. It is easy to see intuitively that the correct mapping is:

```
Id<T1>.Map<T1>(Func<T, T1> f) => new Id<T1>(f(Value))
```

Because if we use this map method with the squaring function `x=>x*x` we will get as a result, an `Id(4)`:

```
var square = x=>x*x;
Id(2).Map(square) === Id(square(2))
```

## Discussion on the Types

Usually when one looks at the map method in the context of Pure functional languages is represented - **map**:  $(a \rightarrow b) \rightarrow f(a) \rightarrow f(b)$  where **f** is a Functor and **a**, **b** are types - (eg. [Haskell](#), [Scala\[Cats\]](#), [Sanctuary.js](#), [Ramda.js](#) ).

We can get this type if we completely extract the method as a function that acts on the object like below:

```
Func<Id<T>, Id<T1>>.Map<T, T1>(Func<T, T1> f) =>
    (Id<T> @this) =>
        new Id<T1>(f(@this.Value));
```

Here the function map has the type  $(T \rightarrow T1) \rightarrow Id<T> \rightarrow Id<T1>$

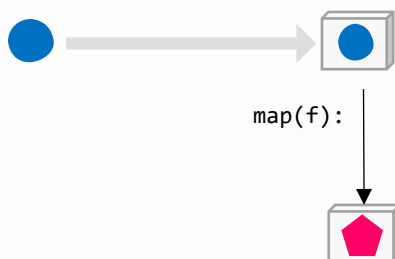
if we rearrange the terms into this form  $Id<T> \rightarrow (T \rightarrow T1) \rightarrow Id<T1>$  (this is the form in [Fantasy land](#) spec) which is the Object oriented version which we can be interpreted in this way : if you give me a function from a to b ( $a \rightarrow b$ ) and I have an `Id<T>`, I can get an `Id<T1>`. The `Id<T>`, in this case is the object (`this`) that contains the method.

## 2.2 Commutative Diagrams

There is one important thing about the mapping function, though, the mapping should reach the same result for  $\text{Id}(4)$  if we take any of the two possible routes to get there. This means **map** (aka lifting of a function from  $\mathbf{C}$  to  $\mathbf{D}$ ) **should preserve the structure of  $\mathbf{C}$** .

1. We could first get the Functor and then map **f**. This is the red path on the diagram.

$$\text{Id}(y) = \text{Id}(x).\text{Map}(f);$$



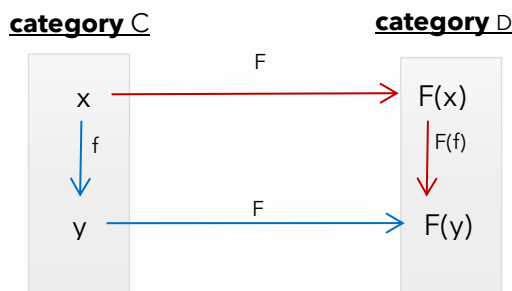
2. Or first lift  $x$  with **f** and then get the Functor.

$$\text{Id}(y) = \text{Id}(f(x));$$



Two objects that were connected with an arrow in category  $\mathbf{C}$ , should also be connected with an arrow in category  $\mathbf{D}$ . And reversely, if there was no connection in  $\mathbf{C}$  there should be no connection in  $\mathbf{D}$ .

When the red and blue paths give the same result, then we say that the diagram **commutes**



Moreover, that means that the lifting of morphisms (aka arrows, aka functions in programming) preserves the structure of the objects in **C**.

In practical day to day programming, commuting diagrams means that **we can exchange the order of operations and still get the same result**. It is not something that happens automatically and is something extremely helpful to have when coding.

## 2.3 Maybe Functor aka Option<T>

### 2.3.1 Maybe Functor Example

Now let us get a client asynchronously with a certain Id from a repository.

```

public class MockClientRepository
{
    List<Client> clients = new List<Client>{
        new Client{Id=1, Name="Jim"},
        new Client{Id=2, Name="John"}
    };

    public Maybe<Client> GetById(int id) =>
        clients.FirstOrDefault(x => x.Id == id);
}
  
```

Run This: [\\_Net Fiddle](#)

Here we have replaced the result of the array filtering:

```

.FirstOrDefault(x => x.Id == id)
  
```

with something that will return a `Maybe<Client>`. If there is no client for a specific id, we should return `None()`. If there is a client, we will return `Some(client)`.

```

public static partial class FunctionalExtensions
{
    public static Maybe<T> FirstOrNone<T>(this List<T> @source, Func<T, bool> predicate)
    {
        var firstOrDefault = @source.FirstOrDefault(predicate);
        if (firstOrDefault != null)
            return new Some<T>(firstOrDefault);
        else
            return new None<T>();
    }
}

```

Now we can write:

```

var repository = new MockClientRepository();

var result = repository.
    GetById(1)
    .MatchWith(pattern:(
        none: () => "Not Found",
        some: (client) => client.Name
    ));

```

Or by using native pattern matching

```

var repository = new MockClientRepository();
var result = repository.
    GetById(1) switch
    {
        None<Client>() => "Not Found",
        Some<Client>(var client) => client.Name,
        _ => throw new NotImplementedException(),
    };

```

Run This: [.Net Fiddle](#)  
 Run This: [ASP.NET MVC Fiddle](#)

Id: int    GetByl    Client.name    switch    string

### SideNote:

You can run a simple ASP.NET MVC [Maybe Functor Example .NET Fiddle](#). Also you can download the [Practical-Functional-CSharp](#) project from Github and run WebApplicationExample.sln the from the **WebApplicationExample** folder.

## 2.3.2 Maybe in Language-ext / Option

The usual name for this functor  $F=1+X$  (disjoint union with a base point) in most functional languages is called **Maybe** but in Language-ext is called **Option** which is another common naming for Maybe that comes closer to the OOP paradigm. Lets say that **Maybe** is the name of the concept and **Option** the implementation. It is the same thing. For the rest of this book **we will use language-ext Option** monad in our code examples.

## 2.3.3 Maybe Functor Example With language-ext Option

Fortunately for us the language-ext have defined **an implicit operator on the Option that does the conversion of null to Option immediately**:

```
public static implicit operator Option<A>(A a);
```

this allows us to write something like the following:

```
public Option<Client> GetById(int id) => clients.SingleOrDefault(x => x.Id == id);
```

We now return an `Option<Client>`

the compiler expects an `Option<Client>` so it uses the operator `Option<A>(A a)` in order to do the conversion on the spot. Now we can write:

```
public class MockClientRepository
{
    List<Client> clients = new List<Client>{
        new Client{Id=1, Name="Jim", },
        new Client{Id=2, Name="John",}
    };

    public Option<Client> GetById(int id) =>
        clients.SingleOrDefault(x => x.Id == id);
}
```

Run This: [.Net Fiddle](#)

We can write again:

```
public string GetNameById(int clientId) =>
    clients
        .GetById(clientId)
        .Map(client => client.Name)
```

```
.Match(Some: (name) => name,
      None: () => "no client");
```

Run This: [.Net Fiddle](#)

## 2.3.4 C# 8. pattern matching Support for Option

Because of the structure of the Option implementation in language-ext library, it's not possible to use `switch` directly on the `Option<>` but fortunately it exposes an `Option<>.Case` property that allow us to use `switch`:

```
public string GetNameById1(int clientId) =>
    clients
    .GetById(clientId)
    .Map(client => client.Name)
    .Case switch
    {
        SomeCase<string>(var name) => name,
        NoneCase<string> { } => "no client",
        _ => throw new NotImplementedException()
    };
```

The Option exposes a `Case` property.

Run This: [.Net Fiddle](#)

The two subtypes that we must match on, are the `SomeCase<>` and `NoneCase<>` and we are going to use this form instead of the `Match` a lot. Both provide the same behaviour, and it comes down to preference.

The only drawback is that **you must write explicitly the Types** of the `SomeCase<>` and `NoneCase<>` because the compiler cannot infer those.

## 2.3.5 Folding Maybe

For the maybe Fold is extremely easy to implement we start with the accumulator and if we have a None then we just return itself or in the case of the Some we apply the reducer to the `accumulator` and the value in order to merge the values:

```
public S Fold<S>(S accumulator, Func<S,T,S> reducer) =>
    this.MatchWith((
        none: () => accumulator,
        some: (v) => reducer(accumulator,v)
    ));
```

Most of the times (almost all) we will prefer the MatchWith to get the value out of the Maybe. We can use Fold like this:

```
public string GetNameById(int clientId) =>
    clients
    .GetById(clientId)
    .Map(client => client.Name)
    .Fold("", (a, name) =>
    {
        return name;
    });
```

Run This: [.Net Fiddle](#)

## 2.3.6 Using the Linq syntax

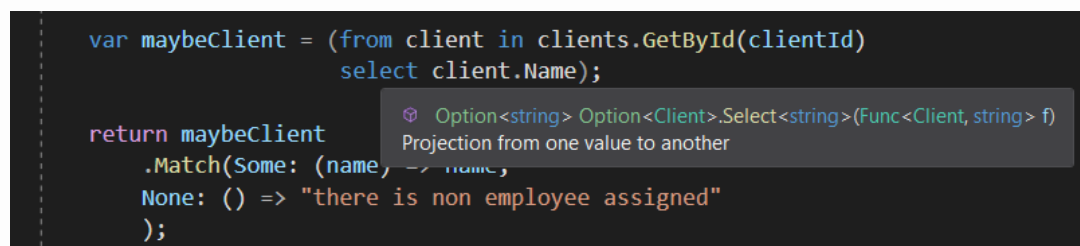
The language-ext library provides a Select method that allows us to use the Linq syntax with the Option:

```
public string GetNameById (int clientId) =>
    (from client in clients.GetById(clientId)
     select client.Name) .Case switch
    {
        SomeCase<string>(var name) => name,
        NoneCase<string> { } => "no client",
        _ => throw new NotImplementedException()
    };
```

Run This: [.Net Fiddle](#)

Run This: [ASP.NET MVC Fiddle](#)

you can see the signature of the select if you place the cursor over the select Keyword.

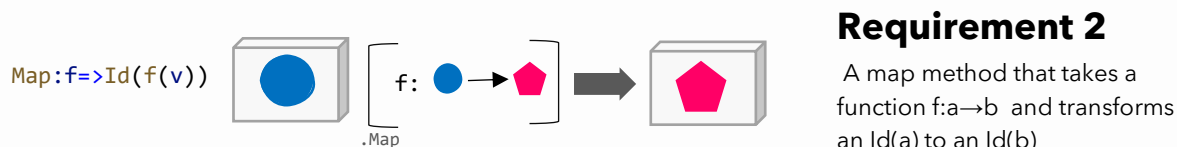


# Monads

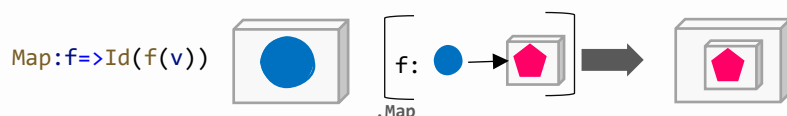
«Wadler tries to appease critics by explaining that "a monad is a monoid in the category of endofunctors, what's the problem ?»  
-Brief, Incomplete and Mostly Wrong History of Programming Languages

## 3 Monads

If you remember in the section "Intro to Functors" we stated that when we use the `map` on a function  $f: A \rightarrow B$  we transform the value  $A$  inside the functor  $F<A>$  to a new type  $F<B>$ .



Sometimes it might happen that the type  $B$  is inside a Functor  $F$  is itself a  $F<B>$ , this means that the function  $f$  might look like this  $f: A \rightarrow F(B)$ . In those situations, after the map we end up with something like  $F<F<B>>$ . A Functor-In-A-Functor situation.



You can see this in the following example

```
Id<Id<int>>IdInId = new Id<int>(5).Map(x => new Id<int>(x + 1));
```

We will now extend our Identity functor to make it into an identity monad.

```
public class Id<T>
{
    public T Value { get; set; }
    public Id(T value) => Value = value;
    public Id<T1> Map<T1>(Func<T, T1> f) => new Id<T1>(f(Value));
}
```

```
public Id<T1> Bind<T1>(Func<T, Id<T1>> f) => f(Value);
}
```

The new thing here is the bind method. `Id<T1> Bind<T1>(Func<T, Id<T1>> f) => f(Value)`

```
Id(5).Bind(x=>Id(x+1)) == Id(6)
```

Run This: [.NET Fiddle](#)

this bind method has a type of **Bind**:  $(T \rightarrow M <T1>) \rightarrow M <T1>$  that takes a function  $T \rightarrow M <T1>$ , which, when provided with the contained value  $v$  it returns a new monad  $M <T1>$  of type  $T1$ .

```
var result = new Id<int>(2)
                .Bind(x => new Id<int>(2)
                .Bind(y => new Id<int>(x + y)));
```

Run This: [.NET Fiddle](#)

Let's put a monad in a monad and Bind with the identity  $x \Rightarrow x$  to get the initial Monad back.

```
var IdInId = new Id<int>(5).Map(x => new Id<int>(x + 1));
var justId = IdInId.Bind(x => x);
```

Run This: [.NET Fiddle](#)

The difference between bind and map in the identity monad is that map passes the lifted value to a monad constructor `Id(f(v))` while bind does not: `f(v)`. In other monads the map will be significantly different than the bind.

The same reasoning goes for all functors, here some other examples leading to the same situation of a Functor in a Functor if we use the Map:

```
New IO(()=>5).Map(x=> new IO(()=> x+1)) === new IO(()=>new IO(()=> 6))
```

```
var a = new[] { 4 }.Select(x => new[] { x + 1 });
var b = new[] { new[] { 4 } };
```

```
new Some<int>(4).Map(x=> new Some<int>(x+1)) === new Some (new Some (5))
```

so its usual to define a Bind in most Functos and thus upgrade them into monads.

### Conventions: Bind and SelectMany

As a convention we will name our mapping method **Bind** nonetheless in some instances we might also use **SelectMany if we need LINQ support**. Map and Select will be used interchangeably in this book. The language-ext library provides **Bind** and **SelectMany** methods aliases

## 3.1 Maybe Monad

Maybe monad is extension of the Maybe Functor. To extend our maybe functor implementation into a Monad we must provide a `Bind` method that combines two Maybe monads into one. **There are 4 different ways to combine the 2 possible states of maybe (some,none)** one can see that a valid implementation would be the following :

```
public abstract class Maybe<T>
{
    public abstract T1 MatchWith<T1>((Func<T1> none, Func<T, T1> some) pattern);

    public Maybe<T1> Bind<T1>(Func<T, Maybe<T1>> f) =>
        this.MatchWith((
            none: () => new None<T1>(),
            some: (v) => f(v)
        ));
}
```

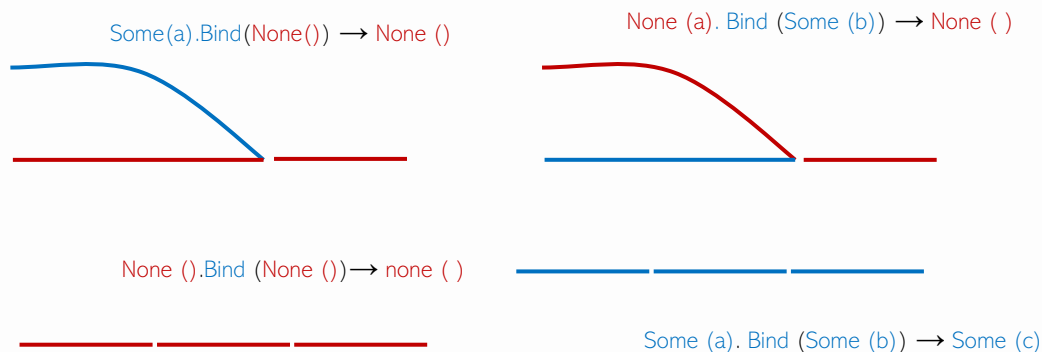
Run This: [.Net Fiddle](#)

Or By using the switch instead:

```
public abstract class Maybe<T>
{
    public Maybe<T1> Bind<T1>(Func<T, Maybe<T1>> f) =>
        this switch
        {
            None<T>() => new None<T1>(),
            Some<T>(var v) => f(v),
            _ => throw new NotImplementedException(),
        };
}
```

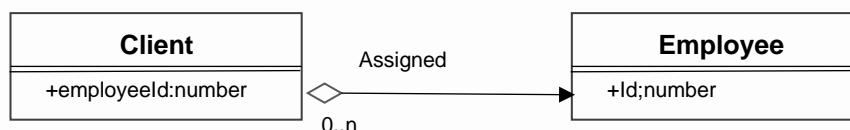
Of course the language-ext library has a `Option<>.Bind` with similar implementation. The only combination that leads to a new `some` (`_`) is when the two paths that both have values are being combined.

```
Some (x). Bind ( None )
None ( ). Bind ( Some )
Some (x). Bind ( Some ) //only this gives a Some result
None ( ). Bind ( None )
```



Let us say that we have this scenario where we have a list of clients and each client is assigned to an employee. Then let us say we want to get the name of the assigned employee for that client.

This is the standard **one-to-many** relationships and the way to model it is to add on the `Client` entity a property that will store the value of the Id of the related employee :



```

public class Client
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int EmployeeId { get; set; }
}
  
```

We use the bind `.Bind(employees.GetById)` to capture the `Client` value from the `Maybe<Client>` that is returned from the `MockClientRepository` and pass it to the `MockEmployeeRepository.GetById` which returns a `Maybe<Employee>`. The final result of the bind is in fact a `Maybe <Employee>`. This is the implementation using the simple our custom `Maybe`

```

public class MockClientRepository
{
    ...
    public Maybe<Client> GetById(int id) => clients.FirstOrNone(x => x.Id == id);
}

public class MockEmployeeRepository
{
    ...
    public Maybe<Employee> GetById(int id) => clients.FirstOrNone(x => x.Id == id);
}

public class Controller{
  
```

```

MockEmployeeRepository employees = new MockEmployeeRepository();
MockClientRepository clients = new MockClientRepository();

public string GetAssignedEmployeeNameById(int clientId) =>
    clients
        .GetById(clientId)
        .Map(client => client.EmployeeId)
        .Bind(employees.GetById)
        .MatchWith((
            none: () => "there is non employee assigned",
            some: (employee) => employee.Name
        ));
}

```

Run This: [.Net Fiddle](#)  
 Run This: [ASP.NET MVC Fiddle](#)  
 Github: [source](#)

### 3.1.1 Using language-ext Option

Rewriting the same using the **language-ext Option** we get

```

public class MockClientRepository
{
    ...

    public Option<Client> GetById(int id) => clients.SingleOrDefault(x => x.Id == id);
}

public class MockEmployeeRepository
{
    ...
    public Option<Employee> GetById(int id) => clients
        .SingleOrDefault(x => x.Id == id);
}

public class Controller
{
    MockEmployeeRepository employees = new MockEmployeeRepository();
    MockClientRepository clients = new MockClientRepository();

    public string GetAssignedEmployeeNameById (int clientId) =>
        clients
            .GetById(clientId)
            .Map(c => c.EmployeeId)
            .Bind(employees.GetById)

```

```

        .Match(
            Some: (employee) => employee.Name,
            None: () => $" No Employee Found"
        );
    }

```

Of course instead of `Match` we could use the `.Case switch`

```

public string GetAssignedEmployeeNameById(int clientId) =>
    clients
        .GetById(clientId)
        .Map(c => c.EmployeeId)
        .Bind(employees.GetById)
        .Case switch
        {
            SomeCase<Employee>(var employee) => employee.Name,
            NoneCase<Employee> { } => "No Employee Found",
            _ => throw new NotImplementedException()
        };

```

Github: [source](#)

# Clean Architecture in .NET

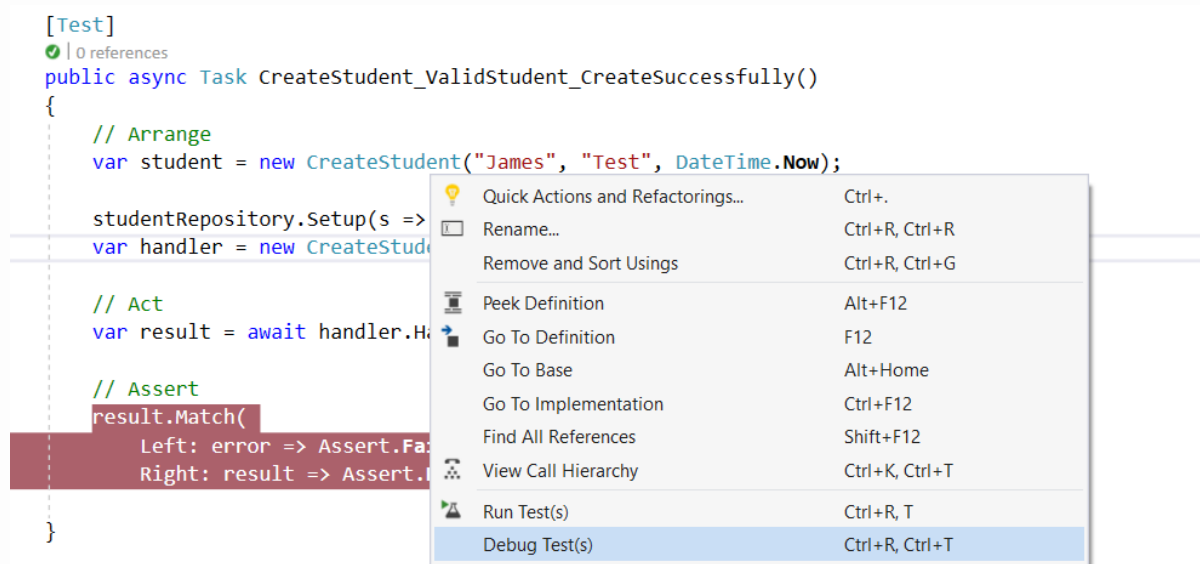
---

In this Section we are going to take a brief look at the possibility of **Functional Software Architecture** and how this can be integrated in the latest trends in **modern Object-oriented software architectures**. This is not a deep dive in Architectural Design but a discussion behind the architecture of the **sample Contoso** web Application in the **language-ext** GitHub project.

## 4 A Clean Functional Architecture Example

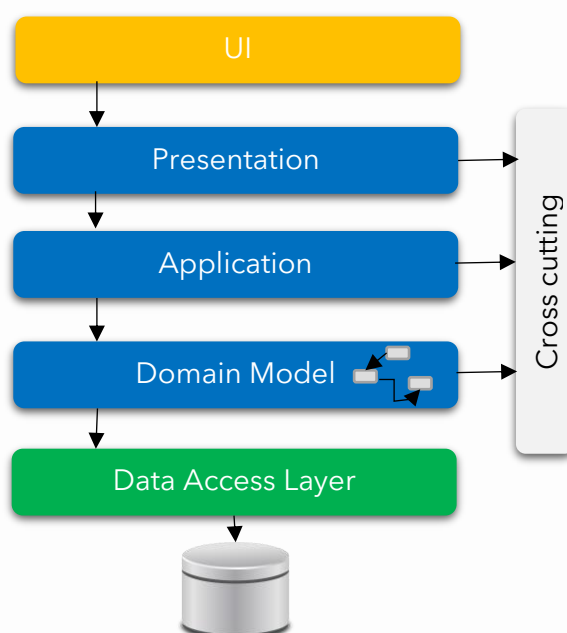
### 4.1 Download and Setup the Project

The project is **headless** and does not have any **Views** implemented. You can mock Http actions using [postman](#) or any other tool, or you might consider building your own front end UI. You can also Debug over the [Tests accompanying](#) the App as an Entry point.



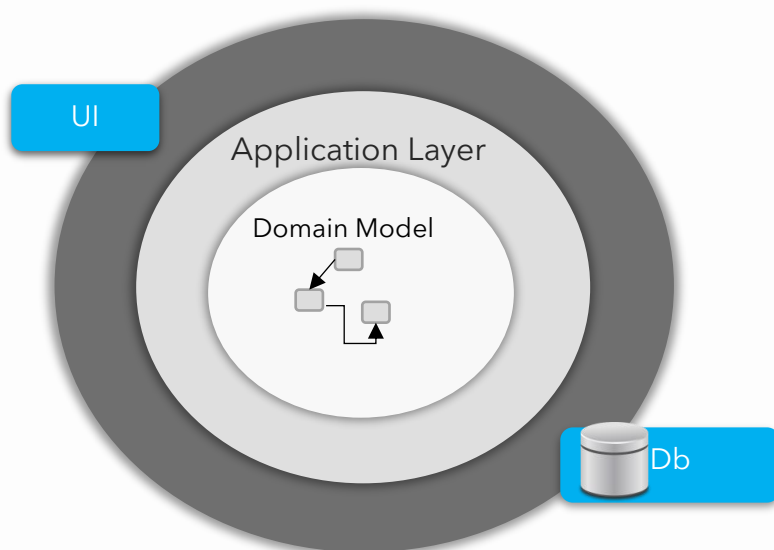
### 4.1.1 Clean Architecture with .NET core

The architecture of web applications depends on the scale and scope of the application. The Previous decade the dominant architecture for small and medium web applications was the **Layered Architecture**. A Layered Architecture, organized the project structure into four main categories: **presentation, application, domain, and infrastructure**.



Under the weight of ideas such as Domain -Driven Design from Eric Evans and Clean Code from "Uncle Bob" - Robert C. Martin the focus of the Architecture became the separation of concerns of the **Domain** from all the technicalities such as technologies, tools and implementation details.

**The domain and its Use Cases** now were placed in the **centre** of the architecture design with minimal dependencies. This led to a series of evolutionary transformations of the Layered structure ( Hexagonal Architecture, Pipes and adapters , DDD etc) leading at to the concept of Clean architecture.



With Clean Architecture, the **Domain** and **Application** layers are at the centre of the design. This is known as the **Core** of the system.

The **Domain** layer contains enterprise logic and types and the **Application** layer contains business logic and types. The difference is that enterprise logic could be shared across many systems, whereas the business logic will typically only be used within this system.

**Core** should not be dependent on data access and other infrastructure concerns, so those dependencies are inverted. This is achieved by adding interfaces or abstractions within **Core** that are implemented by layers outside of **Core**. For example, if you wanted to implement the Repository pattern you would do so by adding an interface within **Core** and adding the implementation within **Infrastructure**.

All dependencies flow inwards, and **Core** has no dependency on any other layer. **Infrastructure** and **Presentation** depend on **Core**, but not on one another.

### Resources:

<https://github.com/maldworth/OnionWebApiStarterKit>

<https://github.com/josecuellar/Onion-DDD-Tooling-DotNET>

[Clean Architecture with ASP.NET Core 2.1 | Jason Taylor](#)

You can also find Visual studio Templates to initialize Projects for example

<https://github.com/jasontaylordev/CleanArchitecture>

## 4.2 A Functional Applications Architecture

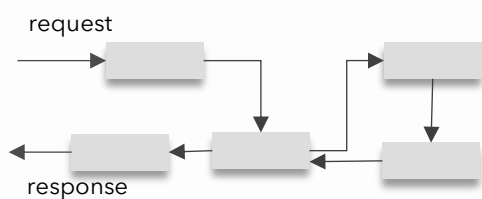
Where does the Functional Programming fit in the larger Architectural view? In the article from Scott Wlaschin "[A primer on functional architecture](#)" he recognizes three principles of functional programming that can be applied to the architectural level:

1. The first is that **functions** are standalone values. In a functional architecture, the basic unit is also a function, but a much larger business-oriented one that I like to call a **workflow**
2. Second, **composition** is the primary way to build systems. Two simple functions can be composed just by connecting the output of one to the input of another
3. functional programmers try to use **pure functions** as much as possible

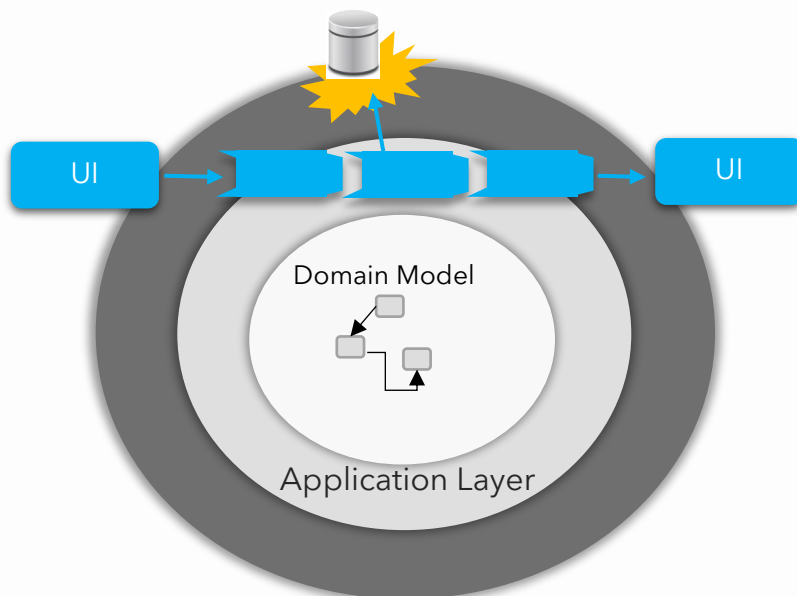
The general idea is that you write as much of your application as possible in an FP style chain computation using the Task, Option, Either, Validation Monads or their combinations and avoiding side effects and coupling. This would lead to the usual pipeline computation style:



Instead of the chaotic object oriented:



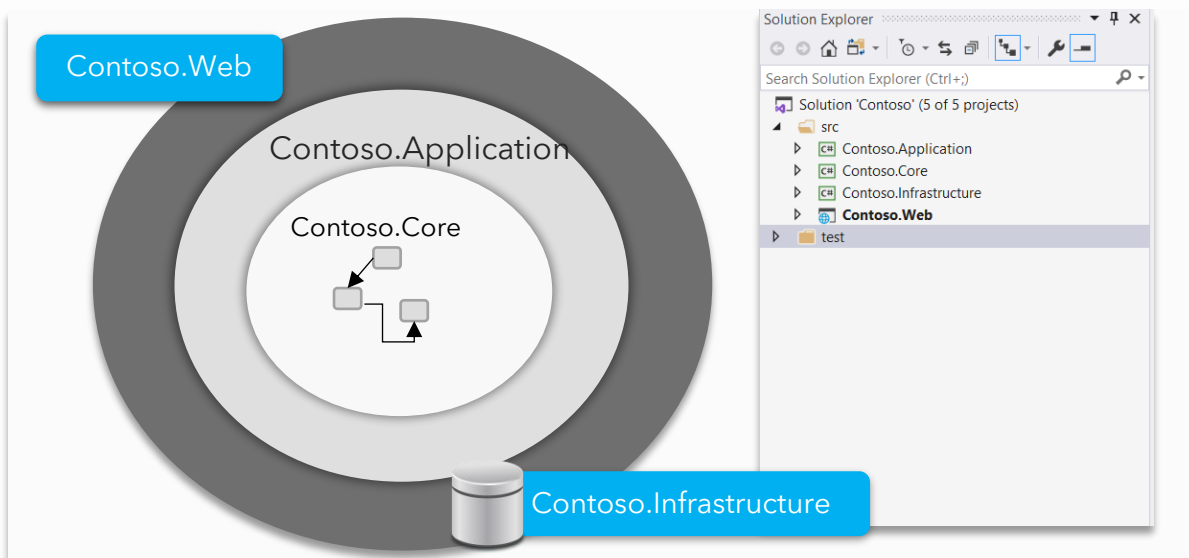
The idea for a functional architecture as laid out by Wlaschin is a pipeline that goes through the layers of the Architecture with minimal coupling and side effects.



### 4.3 The Contoso Clean Architecture with .NET core and language-ext

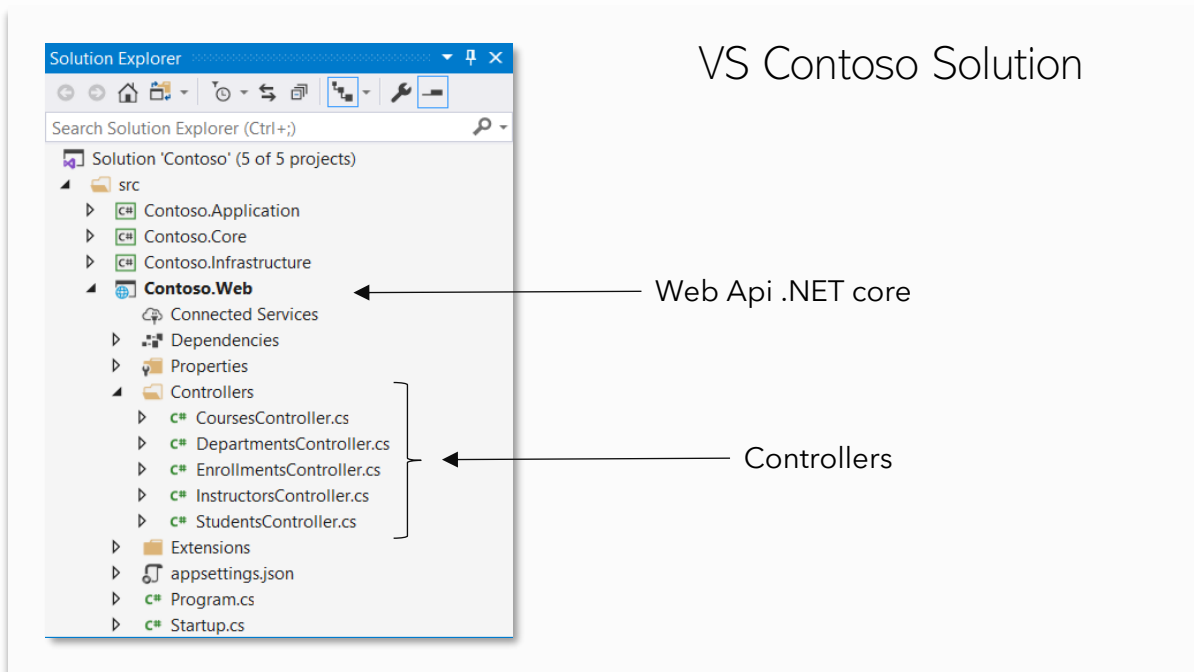
The Contoso application **approximates a Functional architecture Style embedded in a Clean architecture.**

The Contoso application consists of four projects the three of them are class libraries targeting the .NET standard 2.0 framework (Contoso.Application, Contoso.Core, Contoso.Infrastructure) and the "UI" Project named Contoso.Web is a .NET 3.0 Web application project compatible with the rest of the projects



## 4.4 Web Api

This layer contains the code for the Web API endpoint logic including the Controllers. The API project for the solution will have a single responsibility, and that is only to handle the HTTP requests received by the web server and to return the HTTP responses with either success or failure.



We will handle exceptions and errors that have occurred in the Domain or Data projects to effectively communicate with the consumer of APIs. This communication will use HTTP response codes and any data to be returned located in the HTTP response body.

In ASP.NET Core Web API, routing is handled using Attribute Routing. If you need to learn more about Attribute Routing in ASP.NET Core go [here](#). We are also using dependency injection to have the MediatR instance injected into the Controller.

```
[Produces("application/json")]
[Route("api/[controller]")]
[ApiController]
public class CoursesController : ControllerBase
{
    private readonly IMediator _mediator;
    public CoursesController(IMediator mediator) => _mediator = mediator;

    [HttpGet("{courseId}")]
    public Task<IActionResult> Get(int courseId) =>
        _mediator.Send(new GetCourseById(courseId)).ToActionResult();

    [HttpPost]
    public Task<IActionResult> Create(CreateCourse createCourse) =>
        _mediator.Send(createCourse).ToActionResult();

    [HttpPut]
    public Task<IActionResult> Update(UpdateCourse updateCourse) =>
        _mediator.Send(updateCourse).ToActionResult();

    [HttpDelete]
    public Task<IActionResult> Delete(DeleteCourse deleteCourse) =>
        _mediator.Send(deleteCourse).ToActionResult();
}
```

Github: [source](#)