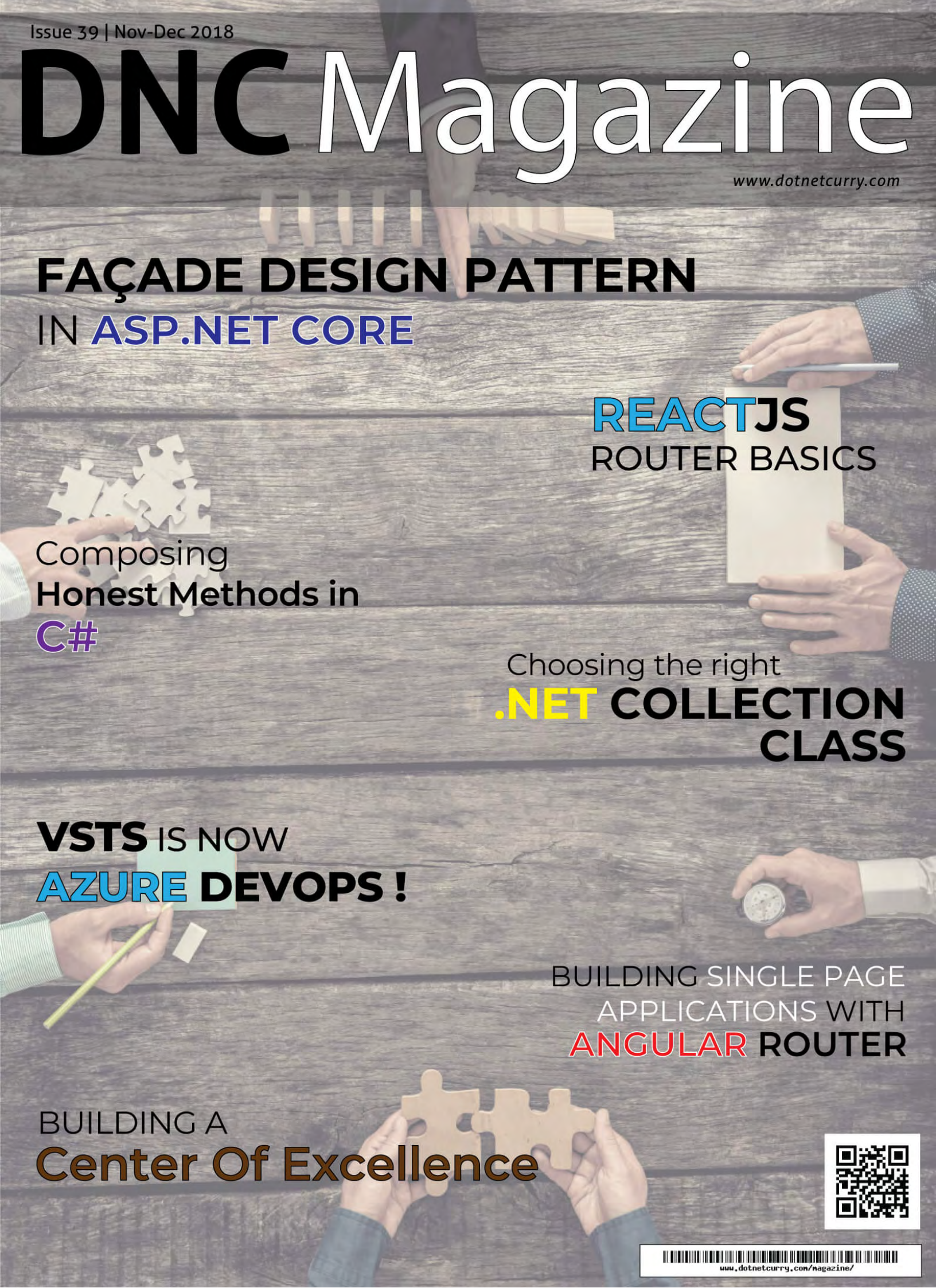
Issue 39 | Nov-Dec 2018

DNC Magazine

www.dotnetcurry.com



FAÇADE DESIGN PATTERN IN ASP.NET CORE

REACTJS ROUTER BASICS

Composing Honest Methods in C#

Choosing the right .NET COLLECTION CLASS

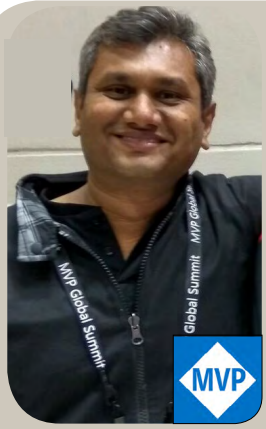
VSTS IS NOW AZURE DEVOPS !

BUILDING SINGLE PAGE APPLICATIONS WITH ANGULAR ROUTER

BUILDING A Center Of Excellence

www.dotnetcurry.com/magazine/

EDITORIAL



Suprotim Agarwal
Editor in Chief

Today, I am excited to announce the pre-order of our eBook ***"The Absolutely Awesome Book on C# and .NET"****.*

The books' author and Microsoft MVP Damir Arh, authored this concise, to-the-point book to strengthen your fundamentals in C# and .NET, to leverage the latest features in C# 7.x and beyond, and to help you crack your next .NET interview. The book is packed with the most useful information and is an absolute must-have for any C# .NET programmer. Sincere thanks to Yacoub Massad for reviewing this book for technical accuracy. Do support our efforts by ***pre-ordering this ebook****, which is expected to be released soon.*

This months' issue of the DNC Magazine is packed with some awesome content. For our C# developers, we have tutorials from Damir and Yacoub on .NET collection classes and composing honest methods. For our JavaScript fans, Keerti and Ravi explain Routing in Angular and ReactJS applications. For our ASP.NET Core developers, first time author Adam Storr explains why the Facade Pattern is still relevant in modern development in ASP.NET Core.

VSTS is now Azure DevOps! Sandeep explains what has changed and why. Finally, Rahul explains what is a Center of Excellence, and presents a blueprint to build one for your organization.

I hope you enjoyed this edition! Reach out to me directly with your comments and feedback at suprotimagarwal@dotnetcurry.com.

THE TEAM

Editor In Chief : Suprotim Agarwal (suprotimagarwal@dotnetcurry.com)

Art Director : Minal Agarwal

Contributing Authors :
Adam Storr
Damir Arh
Keerti Kotaru
Rahul Sahasrabuddhe
Ravi Kiran
Sandeep Chadda
Yacoub Massad

Technical Reviewers :
Damir Arh
Ravi Kiran
Shravan Kumar Kasagoni
Subodh Sohoni
Yacoub Massad

Next Edition : Jan 2019

a2z | Knowledge Visuals

et curry.com

CONTENTS

RAVENDB 4.1

Your Fully Transactional NoSQL Database

The amount of data your organization needs to handle is rising at an ever-increasing rate. We developed RavenDB 4.1 so you can handle this tougher challenge and do it while improving the performance of your application at the same time.

Enjoy top performance while maintaining ACID Guarantees

- 1 million reads/150,000 writes per second on a single node
- Native storage engine designed to soup up performance

All-in-One Database

- Map-reduce and full-text search are part of your database
- No need for plugins

Performs well on smaller servers and older machines

- Unprecedented hardware resource utilization
- Works well on Raspberry Pi, ARM Chips, VM, In-Memory solutions

Easy to Setup and Secure

- Setup wizard gets you started in minutes with top level data encryption for your data

Distributed Counters

- Increment a value without modifying the whole document
- Automatically handle concurrency even when running in a distributed system

Works with SQL Solutions

- Seamlessly send data to SQL databases
- Migrate easily from your current SQL solutions

Expand and Contract Your Cluster on the Fly

- Add new nodes in a click
- Your data cluster is fully transactional
- Assignment failover reassigns tasks from a downed node instantly

The DBAs Dream

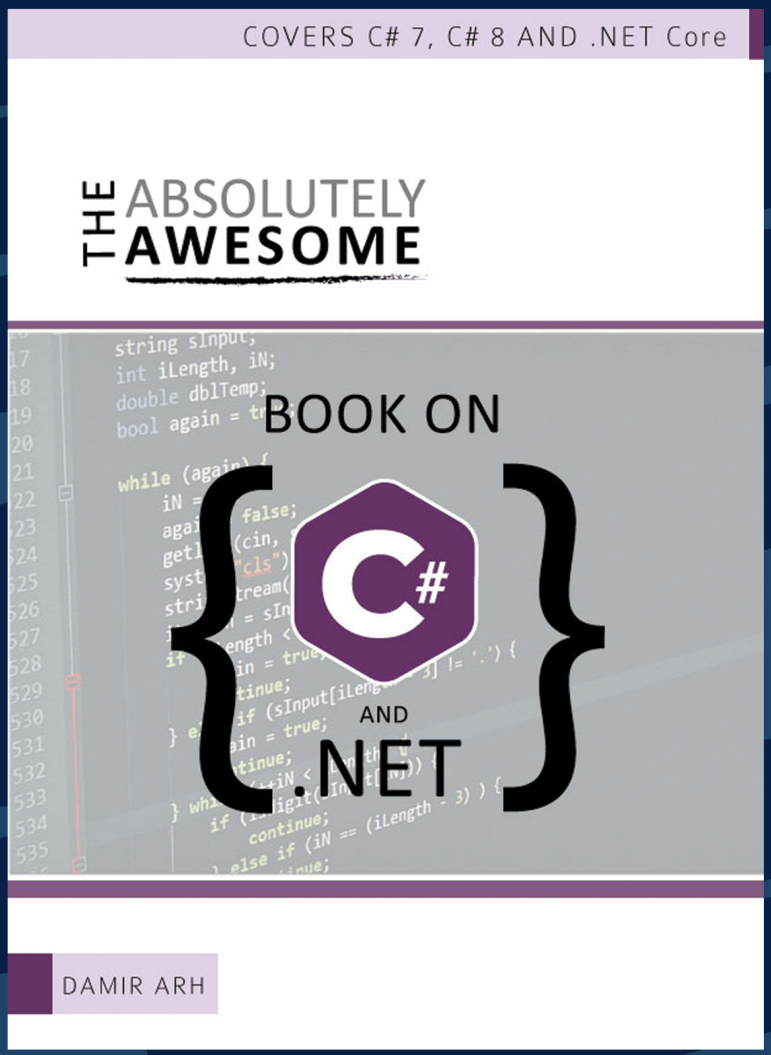
- The most functional GUI on the market, manage your server directly from the browser without doing any complex configuration from the command line

Grab a FREE License

3-node database cluster with GUI interface, 3 cores and 6 GB RAM

www.ravendb.net/free

THERE'S A HOT NEW BOOK IN TOWN!



Features

- ☑ .NET Framework and CLR
- ☑ New features in .NET
- ☑ Type System
- ☑ Generics and Collections
- ☑ C# 6,7 and 8
- ☑ Parallel Programming
- ☑ Async Programming
- ☑ LINQ

It's got it all!

Crack your next .NET Interview

Build a Solid Foundation

Strengthen Concepts

THE ABSOLUTELY AWESOME BOOK ON

C#

PRE ORDER NOW !

Yacoub Massad

COMPOSING HONEST METHODS IN

In this article, I provide a proof of concept in C# for composing honest methods. We will see how to create honest programs out of many honest methods.

Honest Methods - Introduction

In a previous article, Writing Honest Methods in C#, I talked about honest methods.

> *An honest method is one that we can understand by looking at its signature alone, i.e., without reading its body.*

In that article, I talked about honesty for pure methods and honesty for impure methods.

In a nutshell, a ***Pure*** method is a method whose output depends solely on its input arguments and that does not have any side effects. This means that a pure method cannot mutate a parameter, cannot read or mutate global state, cannot read the system timer, cannot read or write file contents, cannot read from or write to the console, etc.

In the same article, I suggested that impure methods should be converted to pure or potentially-pure methods for them to become *honest*.

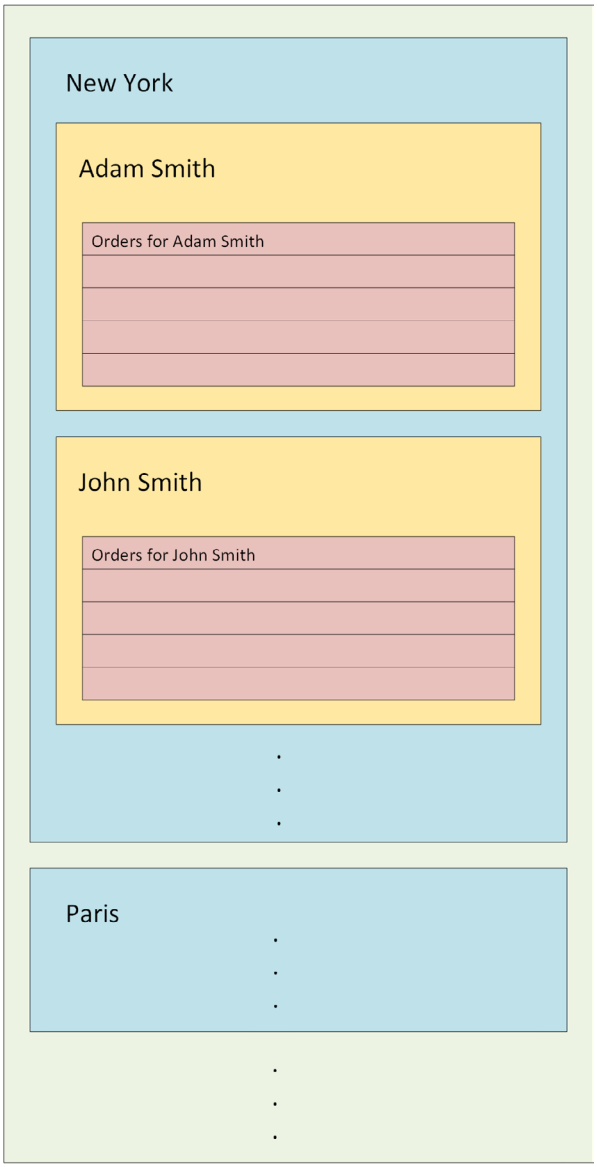
A potentially-pure method is a method whose body is pure, but that invokes functions passed to it as parameters that can be pure or impure (the function parameters types can be delegates for example).

In another article, Writing Pure Code in C#, I discussed examples where impure methods were converted to potentially pure methods. For example, a method that initially invoked `Console.WriteLine` directly, was changed to invoke a delegate of type `Action<string>` passed to it as a parameter. Refer to that article (www.dotnetcurry.com/csharp/1464/pure-code-csharp) for more details.

Towards the end of the Writing Honest Methods in C# article, I talked about a problem related to **making all impure methods in an application, potentially-pure**.

In this article I will describe this problem in more details and then present a proof-of-concept (POC).

The example application



In this article, I will work on a command line application that generates reports based on some data in the database and then saves such a report to a file.

The data is related to customer orders; each customer in the database has orders. The report is partitioned by the city in which the customers live. So, the structure of the report looks like this the one shown in Figure 1:

Figure 1: Report structure

Figure 2 shows how the methods of the application call each other:

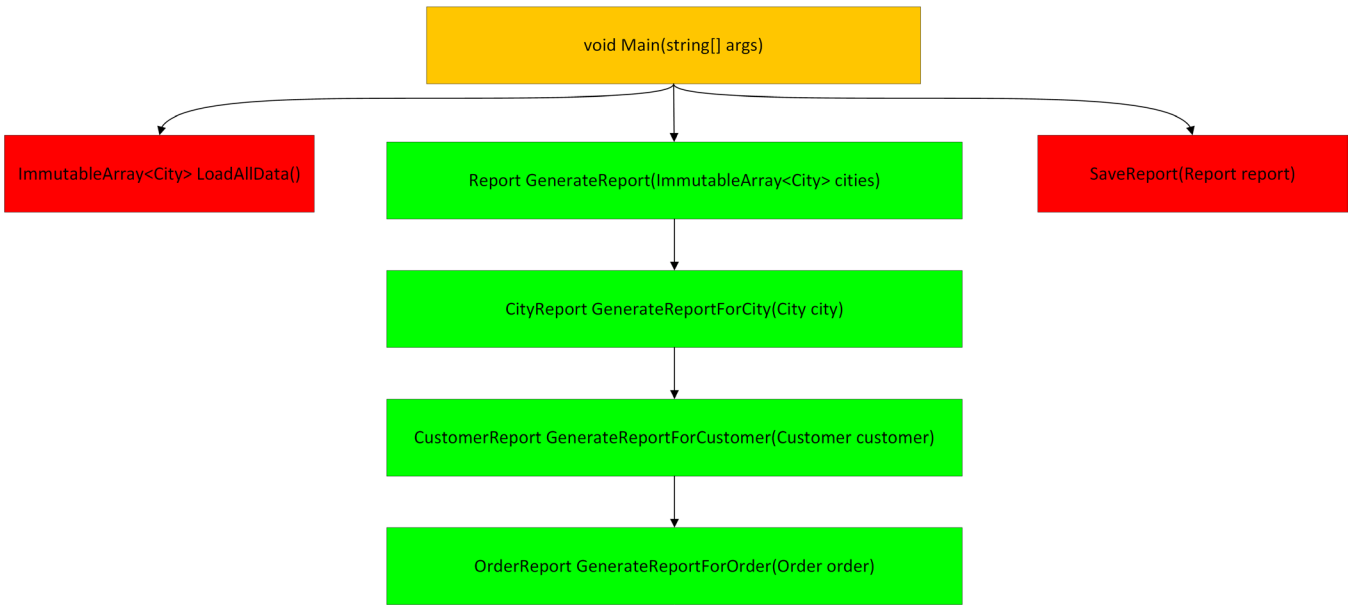


Figure 2: The method call tree

The `Main` method starts by calling the `LoadAllData` method to obtain all the data needed to generate the report from the database. It then calls the `GenerateReport` method passing in the array of `City` objects it got from `LoadAllData`.

The `City` object in this case contains `Customer` objects representing all the customers living in the city. Each `Customer` object contains `Order` objects representing the orders the customer has made.

The `GenerateReport` method invokes the `GenerateReportForCity` method for each `City` object to generate a sub-report for each city and then it aggregates all these sub-reports to generate the full report. We can also expect the `GenerateReport` method to add other information to the report such as the total number of cities, total number of customers, total number of orders, etc.

The `GenerateReportForCity` method does something similar, it calls `GenerateReportForCustomer` for each customer and then aggregates the results into a `CityReport` object. The `GenerateReportForCustomer` method also does a similar job. It calls `GenerateReportForOrder` for each `Order` object.

Of course, we can expect each of these methods to organize the sub-reports in a different way. For example, `GenerateReportForCity` might sort the customer reports by customer name, while the `GenerateReportForCustomer` method might sort the order reports by date or total amount. Also, the additional data these methods add to the aggregated report, is expected to be different.

The `GenerateReportForOrder` method returns an `OrderReport` object for the `Order` object it gets. We can expect this returned report to contain useful details about the order such as the products ordered, total price for each product, the order total price, etc.

In the figure, I colored the `Main` method in orange, the `LoadAllData` and the `SaveReport` methods in red, and the other methods in green. This is the same coloring scheme I used in the Writing Honest Methods in C# article.

In summary, I use **red to indicate that a method is impure, green to indicate that a method is pure, and orange to indicate that although the method has a pure body, it calls other methods that are impure**. See the mentioned article for more details.

The `LoadAllData` method is impure because it communicates with the database. The `SaveReport` method is impure because it writes to the file system. The `Main` method is impure because it invokes `LoadAllData` and `SaveReport`. The other methods contain pure logic.

I have created a sample project to demonstrate the ideas in this article. You can find the source code here: https://github.com/ymassad/ComposingHonestFunctionsExamples

The solution contains ten projects. For now, look at the `ReportGenerator` project.

The sample application does not read from the database, instead the `LoadAllData` method returns some constant data. Also, the `SaveReport` method simply outputs the report to the console. The contents of each *Report object generated by the corresponding Generate* method is simply a string containing a simplistic description of the object passed to the Generate* method.

I did all this to simplify the sample application. In the `LoadAllData` method, I intentionally increment a static field called `DummyState` to make the method impure. This is relevant if you decide to use the PurityAnalyzer extension to analyze the application code.

For more information about this extension, see the Writing Pure Code in C# article.

Here is the code for the Main method:

```
static void Main(string[] args)
{
  var cities = LoadAllData();

  var report = GenerateReport(cities);

  SaveReport(report);
}
```

Listing 1: The Main method

In the Writing Honest Methods in C# article, I mentioned the concept of the impure/pure/impure sandwich. We can see such a concept applied here in the sample application.

First, the impure `LoadAllData` method is invoked to get all the data needed to generate the report. Then, the pure `GenerateReport` method is invoked to generate the report. Then, the impure `SaveReport` method is invoked to save the report.

Great! So we have separated the pure part from the impure parts in the best way possible!

Well, not really!!

Your manager comes to your office and tells you that some client is having trouble running the application.

After investigation, you discover that the client data set is huge and that the call to `LoadAllData` fails because of this huge data set.

To fix the issue, you decide to make the `LoadAllData` method not include the `Order` objects. You defer loading orders for each customer from the database to the `GenerateReportForCustomer` method. See the updated application code in the ReportGenerator2 project.
Here is how the method call tree looks like now:

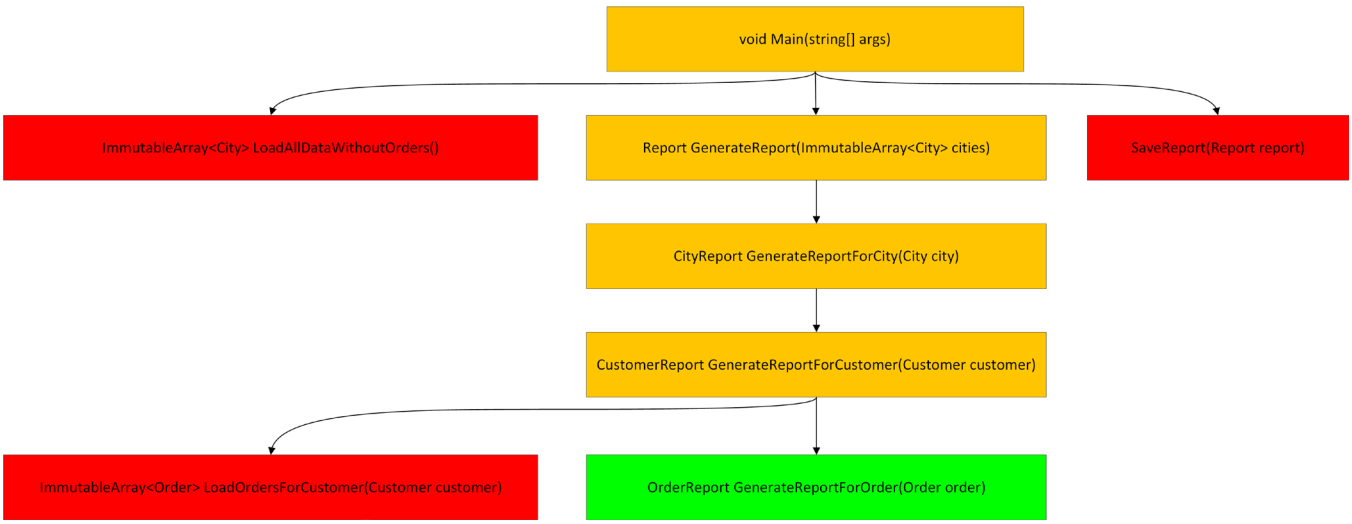


Figure 3: Method call tree after making the GenerateReportForCustomer method invoke an impure method to obtain customer orders

The `LoadAllData` method is now renamed to `LoadAllDataWithoutOrders.` Also, the `GenerateReportForCustomer` method calls the impure `LoadOrdersForCustomer` method to get the `Order` objects for the customer from the database. To make things simple, the `LoadOrdersForCustomer` method in the sample application does not speak to the database. Instead, it returns constant data.

Notice how I changed the colors of three Generate* methods to orange. The bodies of these methods are still pure. However, they are impure because they end up calling the `LoadOrdersForCustomer` method, which is impure.

Now, only the `GenerateReportForOrder` method is pure. You are no longer happy! Three Generate* methods are no longer honest!

How to fix this?

You decide to make the `GenerateReportForCustomer` method potentially-pure by making it invoke a `getOrdersForCustomer` function parameter (which is a new parameter added to this method) instead of invoking the impure `LoadOrdersForCustomer` method directly. Also, because you want to make `GenerateReportForCity` and `GenerateReport` potentially-pure, you make them take a `getOrdersForCustomer` parameter which they simply pass down.

The updated code is in the ReportGenerator3 project.

Figure 4 shows how the method call tree looks like:

Figure 4: Method call tree after making the three Generate methods potentially-pure

I changed the color of the three Generate* methods back to green. These methods are potentially-pure, not pure. But because potentially-pure methods are honest about their impurities, it makes sense to look at them in a positive way like this. I talk about this in more details in the Writing Honest Methods in C# article.

When calling the `GenerateReport` method, the `Main` method passes the impure `LoadOrdersForCustomer` method as an argument for the `getOrdersForCustomer` parameter. In some sense, we have brought purity back to the Generate* methods.

Now comes the problem I described at the end of the Writing Honest Methods in C# article. Three Generate* methods have to pass `getOrdersForCustomer` down the call stack even if they don't use it directly.

The sample application is a simple one. A real application could have call stacks that have 20 methods in them. Also, the number of parameters that intermediate methods have to pass to the lower methods, can be large.

For example, consider that a client wants to control aspects of order report generation by using command line parameters. You decide to create an `OrderReportFormattingSettings` class to model such aspects. You create an instance of this class in the `Main` method, populate it based on the command line arguments, and then pass it through the Generate* methods down to `GenerateReportForOrder`. The new code is in

the ReportGenerator4 project.

Figure 5 shows the method call tree:

Figure 5: Adding the OrderReportFormattingSettings parameter to the Generate* methods

So, whenever a lower-level method requires a new parameter, either some settings related parameter, or some impure functionality, we need to update all the methods from the `Main` method down to this method so that they pass the new parameter.

This is a maintainability issue.

What is the solution?

Dependency Injection

Well, the solution is Dependency Injection.

Dependency Injection allows us to fix some parameters at composition time and have some parameters be specified at runtime. Consider this class:

```
public sealed class SomeClass
{
  public readonly int valueToAdd;
  public SomeClass(int valueToAdd)
  {
    this.valueToAdd = valueToAdd;
  }
  public int AddSome(int value) => value + valueToAdd;
}
```

Listing 2: A class that takes one parameter at composition time and one at runtime

When creating an instance of this class, specify the value for `valueToAdd`. Every time you invoke `AddSome`, you can pass a different value for the value parameter. However, `valueToAdd` is fixed because it is set at composition time.

We can convert the four Generate* methods into classes and use Dependency Injection to have the `orderReportFormattingSettings` and `getOrdersForCustomer` parameters fixed at composition time, i.e., in the Main method.

The new code is in the ReportGenerator5 project.

Here are how the *Generator classes composed and used in the `Main` method:

```
var orderReportGenerator =
  new OrderReportGenerator(
    orderReportFormattingSettings);

var customerReportGenerator =
  new CustomerReportGenerator(
    orderReportGenerator,
    LoadOrdersForCustomer);

var cityReportGenerator =
  new CityReportGenerator(
    customerReportGenerator);

var reportGenerator =
  new ReportGenerator(
    cityReportGenerator);

var report = reportGenerator.Generate(cities);
```

Listing 3: The four *Generator objects are composed in the Main method

Notice how only the `OrderReportGenerator` class has a dependency (via a constructor parameter) on `orderReportFormattingSettings`. Also, only the `CustomerReportGenerator` class has a dependency on `LoadOrdersForCustomer`.

Notice also how each *Generator object (except for `orderReportGenerator`) is given an instance of the lower-level *Generator class at composition time.

If in the future, the `OrderReportGenerator` class requires more dependencies, we don't have to change all the classes that use it (directly or indirectly). We only need to change it and then provide the required dependency in the Composition Root (the `Main` method).

This is highly maintainable.

But what about honesty? Is the code honest?

Let's start with the *Generator classes themselves. Consider the ReportGenerator class for example. If we look at the signature of the `Generate` method inside it, it only takes in an array of cities. We cannot know from the `Generate` method signature alone that it will invoke the `CityReportGenerator.Generate` method.

Still, one can argue that by looking at the signature of the constructor of `ReportGenerator`, we know the `ReportGenerator` class has a dependency on `CityReportGenerator`. Since both classes have a single method, we can assume that the `Generate` method has a dependency on the `CityReportGenerator.Generate` method.

But that is not good enough.

We could not know that the `getOrdersForCustomer` parameter of CustomerReportGenerator's constructor will eventually be called by the `ReportGenerator.Generate` method.

One could argue though that since the *Generator classes depend on each other, we can navigate the dependency tree and eventually know about `getOrdersForCustomer` (and also `orderReportFormattingSettings`). That is:

1. From the signature of the constructor of `ReportGenerator`, we know about `CityReportGenerator`.
2. From the signature of the constructor of `CityReportGenerator`, we know about `CustomerReportGenerator`.
3. From the signature of the constructor of `CustomerReportGenerator`, we know about `getOrdersForCustomer`.
4. From the signature of the constructor of `CustomerReportGenerator`, we know also about `OrderReportGenerator`.
5. From the signature of the constructor of `OrderReportGenerator`, we know about `orderReportFormattingSettings`.

This is not as simple as looking into a single method's signature, but it is doable.

Usually, to get the full benefits of Dependency Injection, we program to interfaces.

For example, instead of having `ReportGenerator` depend on `CityReportGenerator` directly, we have it depend on `ICityReportGenerator`. This would be an interface implemented by `CityReportGenerator`. Usually this is done to enable the customization of the `CityReportGenerator` class without modifying it. For example, we can create a decorator for `ICityReportGenerator`, say `CityReportGeneratorLoggingDecorator`, that logs to some file the fact that a specific city-level report is being generated.

When that happens, we can no longer navigate the concrete dependencies using the signature of the constructors. When we look at the constructor of the updated ReportGenerator class for example, we see `ICityReportGenerator`, not `CityReportGenerator`.

The new code is in the ReportGenerator6 project.

Is the ReportGenerator class honest now?

The `ReportGenerator.Generate` method has a pure body, and the constructor of the ReportGenerator class tells us that the `Generate` method invokes `ICityReportGenerator.Generate` which could be impure. In this sense, we can say the `ReportGenerator` class is honest.

Looking at this from a different angle, the `ReportGenerator` class does not tell us what concrete implementation would be behind `ICityReportGenerator`. This is no longer the job of the `ReportGenerator` class. The Composition Root (the Main method) is the one responsible for this.

This means we should focus on the Composition Root for our questions about honesty.

Let's look at it again:

```
var orderReportGenerator =
  new OrderReportGenerator(
    orderReportFormattingSettings);

var customerReportGenerator =
  new CustomerReportGenerator(
    orderReportGenerator,
    LoadOrdersForCustomer);

var cityReportGenerator =
  new CityReportGenerator(
    customerReportGenerator);

var reportGenerator =
  new ReportGenerator(
    cityReportGenerator);

var report = reportGenerator.Generate(cities);
```

Listing 4: The four *Generator objects are composed in the Main method

The `reportGenerator` variable holds an instance of the `ReportGenerator` class.

Let's ask the following question: **is `reportGenerator.Generate` honest?**

Well, from the signature of this method:

```
Report Generate(ImmutableArray<City> cities)
```

...we can't know it will eventually invoke `LoadOrdersForCustomer`. However, we can navigate the constructor of `ReportGenerator` to find that it depends on `ICityReportGenerator`.

Now, looking back at the `Main` method, we know that an instance of the `CityReportGenerator` class will be used as `ICityReportGenerator` inside the `ReportGenerator` instance. We can continue the same navigation process we did before and use the code in the `Main` method to know which concrete implementation is used behind which interface. Doing this to the end, we will know that `reportGenerator.Generate` eventually calls `LoadOrdersForCustomer`.

It was hard before. And now it is even harder. I don't think we can consider `reportGenerator.Generate` to be honest.

Currently, the dependency tree has a depth of four (we have four *Generator classes). What happens when we have a depth of 20 or 30?

We need something better.

Before discussing the solution, I will talk about a way to do Dependency Injection without using classes.

Partial Invocation

Partial invocation is a functional programming technique that allows us to invoke a function with only some of its parameters giving us back another function, that takes the rest of the parameters.

For example, assume we have the following function, represented by a C# Func:

```
Func<int, int, int, int> calculate = (x, y, z) => x * y + z;
```

This function takes three integers as parameters and returns another integer as a result based on the specified equation.

Consider the following code:

```
Func<int, int, int> calculate1 = (y, z) => calculate(1, y ,z);
```

This new function takes two integers as parameters and invokes the calculate function passing a constant value of 1 as the first argument, and then the values for y and z as the next two arguments.

In C#, to create calculate1, I had to manually define it to invoke `calculate`. However, in other languages, there is a built-in support for doing this.

For instance, imagine that we could do something like this:

```
var calculate1 = calculate(1);
```

In C#, this code will not compile. But in F# for example, a similar code would compile, and the result would be like how we defined calculate1 the first time.

Editorial Note: *Here's an article written by Damir Functional Programming (F#) for C# Developers that talks about some functional approaches you are already using in C#.*

Let's look at the `SomeClass` class again:

```
public sealed class SomeClass
{
    public readonly int valueToAdd;

    public SomeClass(int valueToAdd)
    {
        this.valueToAdd = valueToAdd;
```

```
    }

    public int AddSome(int value) => value + valueToAdd;
}
```

Listing 5: A class that takes one parameter at composition time and one at runtime

This class can be replaced by the following function:

```
Func<int, int, int> AddSome = (valueToAdd, value) => value + valueToAdd;
```

We can call it directly like this:

```
var result = AddSome(2, 1);
```

But we can also partially invoke it (in an imaginary version of C#) like this:

```
var AddSome2 = AddSome(2);
```

And then give `AddSome2` to any consumer who needs to call `SomeClass.AddSome`. This would be like constructing `SomeClass` like this:

```
var AddSome2 = new SomeClass(2);
```

The difference is that partial invocation is more flexible. Given a function, we can choose which parameters to fix at composition time and which parameters we need to keep for the callers to specify at runtime.

The same function can be partially invoked in different ways, e.g. different number of fixed parameters at composition time, and then the results can be given to different consumers.

I have been working on a project to enable partial invocation in C#. I will show you sample code and then I will discuss it. See the code in the ReportGenerator7 project.

I have removed the *Generator classes and the corresponding interfaces. I have created a static class called `ReportingModuleFunctions` that contains public static methods that have the following signatures:

```
public static Report GenerateReport(
    Func<City, CityReport> generateCityReport,
    ImmutableArray<City> cities)

public static CityReport GenerateReportForCity(
    Func<Customer, CustomerReport> generateCustomerReport,
    City city)
public static CustomerReport GenerateReportForCustomer(
    Func<Customer, ImmutableArray<Order>> getOrdersForCustomer,
    Func<Order, OrderReport> generateOrderReport,
    Customer customer)

public static OrderReport GenerateReportForOrder(
    OrderReportFormattingSettings orderReportFormattingSettings,
    Order order)
```

Listing 6: The four Generate* methods

These methods are similar to the methods in the ReportGenerator4 project. The difference though is that these methods do not call each other directly.

For example, the `GenerateReport` method does not call `GenerateReportForCity` directly, instead it takes a parameter of type `Func<City, CityReport>` that it uses to generate the sub-report for each city.

These four static methods are the functional equivalent of the four classes used in the ReportGenerator6 project.

For example, the `GenerateReport` method takes a parameter of type `Func<City, CityReport>`, and the `ReportGenerator` class constructor takes a parameter of type `ICityReportGenerator` which has a single method that takes a `City` object and returns a `CityReport` object. Also, the `GenerateReport` method takes a parameter of type `ImmutableArray<City>`, and the `Generate` method in the `ReportGenerator` class takes a similar parameter.

Like we used Dependency Injection to connect the four classes together in the `Main` method in the ReportGenerator6 project, in the ReportGenerator7 project we use Partial Invocation to connect the four functions together. Here is how the Main method looks like:

```
var generateReportForOrder =
    ReportingModule.GenerateReportForOrder()
        .PartiallyInvoke(orderReportFormattingSettings);

var generateReportForCustomer =
    ReportingModule.GenerateReportForCustomer()
        .PartiallyInvoke(
            generateReportForOrder,
            DatabaseModule.LoadOrdersForCustomer());

var generateReportForCity =
    ReportingModule.GenerateReportForCity()
        .PartiallyInvoke(generateReportForCustomer);

var generateReport =
    ReportingModule.GenerateReport()
        .PartiallyInvoke(generateReportForCity);
```

Listing 7: The four Generate* methods are composed together

Listing 7 is very similar to Listing 3. For example, compare how we create the `OrderReportGenerator` object in Listing 3, to how we obtain the `generateReportForOrder` function. In Listing 3, we use the constructor to specify `orderReportFormattingSettings`, and in Listing 7, we use a special method, the `PartiallyInvoke` method, to specify it. The same goes for the other objects/functions.

ReportGenerator7 is equivalent to ReportGenerator6. We are just using functions instead of objects.

The analysis we did for method honesty for the code in the ReportGenerator6 project, applies here too. We still haven't found a way to make the functions composed in the Main method, honest.

Before discussing one solution, let's first talk about the `PartiallyInvoke` method and the `ReportingModule` class.

The InterfaceMappingHelper Visual Studio extension

The `PartiallyInvoke` methods used in the Main method are auto generated methods. If you look in the Solution Explorer window under the Program.cs file in the ReportGenerator7 project, you will see a file called Program.Mapping.cs. This file was auto generated by a Visual Studio extension I developed called InterfaceMappingHelper.

I talked about some features of this extension in the Aspect Oriented Programming in C# via Functions article. The features I used in the ReportGenerator7 project are different ones though. If you look at the properties of the Program.cs file (via the Properties window), you will find that the Custom Tool property has the value of MappingCodeGenerator. This tool is part of the InterfaceMappingHelper extension. For more information about custom tools in Visual Studio, see this reference: https://docs.microsoft.com/en-us/visualstudio/extensibility/internals/custom-tools?view=vs-2017

When we save the Program.cs file, the MappingCodeGenerator Custom Tool runs. It will look for calls to `PartiallyInvoke` (a dummy extension method inside the `FunctionExtensionMethods` class) inside Program.cs and generate a real `PartiallyInvoke` method inside Program.Mapping.cs.

But why do we need to have these methods generated? Why not include them in some library?

The reason is that the number of all the possible variations of the `PartiallyInvoke` method is huge.

For example, say you can have a function with 20 input parameters, and decide to partially invoke the function with 5 arguments. You can choose to pass arguments for parameters 1, 4, 6, 13, and 19. I hope you can now imagine how the number of possible overloads of `PartiallyInvoke` can become very large.

The ReportingModuleFunctions.cs and the DatabaseModuleFunctions.cs files also have the MappingCodeGenerator set as the value for their Custom Tool property.

When the Custom Tool finds a static class whose name ends with "Functions", it generates a corresponding class in the *.Mapping.cs file whose name is the same as the static class but without "Functions". For example, a ReportingModule class will be generated for ReportingModuleFunctions.

For each public method in the *Functions class, say MethodX, a public method will be generated in the corresponding class in the *.Mapping.cs file. Such generated method returns a function which simply calls MethodX. This is done for convenience because in C#, there is no convenient way to convert a static method to a function. For example, if you have a static method like this:

```
public static string FormatDate(DateTime date)
```

There is no convenient way to obtain a `Func<DateTime, string>` from this method. For example, there is no way to do something like this:

```
var formatDateFunc = FormatDate.ToFunc();
```

We can do it like this:

```
var formatDateFunc = new Func<DateTime, string>(FormatDate);
```

But this is not convenient because we have to specify the parameter types and the return type of the `FormatDate` method.

To see how the Custom Tool helps, consider that the programmer creates a static method of the following signature:

```
public static OrderReport GenerateReportForOrder(
  OrderReportFormattingSettings orderReportFormattingSettings,
  Order order)
```

And now the generated `ReportingModule.GenerateReportForOrder` method returns a function of `type Func<(OrderReportFormattingSettings orderReportFormattingSettings, Order order), OrderReport>` representing the static method.

Notice how the two input arguments are put inside a tuple. For more information about this, please see the Aspect Oriented Programming in C# via Functions article.

Note: *The InterfaceMappingHelper extension by default understands an interface called* `IFunction` *and not the* `Func` *delegate. It can however be configured to use the System.Func<T,TResult> delegate. The settings can be found in the Visual Studio menu > Tools > Options > Interface Mapping Helper.*

Figure 6 shows how this looks on my machine:

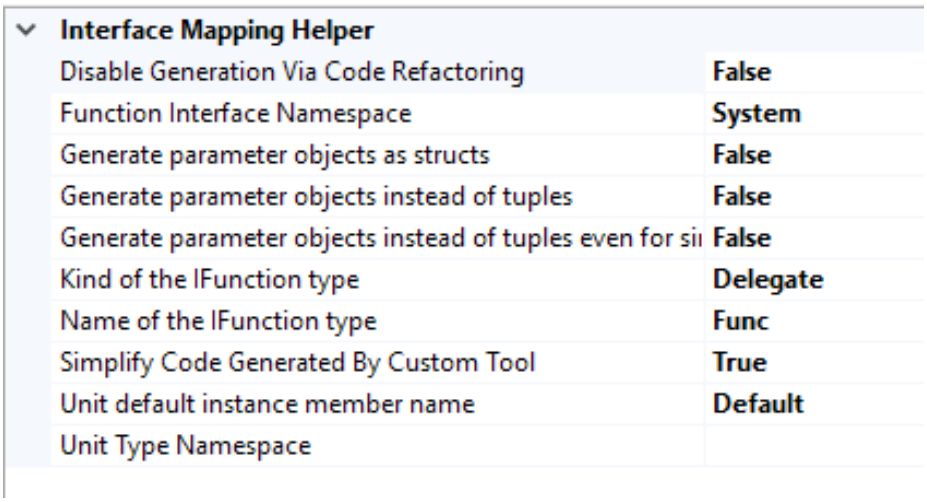


Figure 6: Modifying the settings of the InterfaceMappingHelper extension to make it use the Func<T,TResult> delegate instead of the IFunction<TInput, TOutput> interface

Honest Dependency Injection

Let's start by looking at the `Main` method in the ReportGenerator8 project.

```
var generateReportForOrder =
    ReportingModule.GenerateReportForOrder();

var generateReportForCustomer =
    ReportingModule.GenerateReportForCustomer()
        .HonestlyInject(
            generateReportForOrder);

var generateReportForCity =
```

```
    ReportingModule.GenerateReportForCity()
        .HonestlyInject(generateReportForCustomer);

var generateReportHonest =
    ReportingModule.GenerateReport()
        .HonestlyInject(generateReportForCity);

var generateReport =
    generateReportHonest
        .PartiallyInvoke(
            orderReportFormattingSettings,
            DatabaseModule.LoadOrdersForCustomer());

var report = generateReport(cities);
```

Listing 8: Using the HonestlyInject method in the Main method

Only the `Main` method is different between the ReportGenerator7 and ReportGenerator8 projects. Note the following differences in the `Main` method (compare Listing 7 to Listing 8):

In Listing 7, the `PartiallyInvoke` method is used to inject functions into other functions, while in Listing 8, a method called `HonestlyInject` is used.

In Listing 7, `orderReportFormattingSettings` is injected into `ReportingModule.GenerateReportForOrder` immediately. That is, before injecting `generateReportForOrder` into `ReportingModule.GenerateReportForCustomer`. In Listing 8, `orderReportFormattingSettings` is used at a later stage; it is injected into `generateReportHonest` to get `generateReport`.

The same thing is true for `DatabaseModule.LoadOrdersForCustomer`. In Listing 7, it is injected into `ReportingModule.GenerateReportForCustomer` immediately. While in Listing 8, it is injected at a later stage into `generateReportHonest` to get `generateReport`.

If you hover over the `generateReportHonest` variable, you will see it has the following type:

```
Func<
    (
    ImmutableArray<City>,
    Func<Customer, ImmutableArray<Order>>,
    OrderReportFormattingSettings
    ),
    Report>
```

Listing 9: The type of the generateReportHonest variable

`generateReportHonest` is a function that takes:

1. an array of cities
2. a function that take a customer and returns an array of orders
3. Order report formatting settings

Basically, the `HonestlyInject` method allows us to inject one function, say function 1, into another function, say function 2, even if function 1 doesn't exactly match the dependency required by function 2.

Consider for example the `ReportingModule.GenerateReportForCustomer` function. It has the following signature:

```
Func<
    (
    Func<Customer, ImmutableArray<Order>> getOrdersForCustomer,
    Func<Order, OrderReport> generateOrderReport,
    Customer customer
    ),
    CustomerReport>
```

Listing 10: Signature of the GenerateReportForCustomer function

This function has a dependency of type `Func<Order, OrderReport>` which is a function that can generate a sub-report for an order object. We want to inject `generateReportForOrder` for this dependency, but the `generateReportForOrder` function has the following signature:

```
Func<
    (
    OrderReportFormattingSettings orderReportFormattingSettings,
    Order order
    ),
    OrderReport>
```

Listing 11: The signature of the generateReportForOrder function

It doesn't exactly match the dependency required by `ReportingModule.GenerateReportForCustomer`. In the ReportGenerator7 project, we had to first partially invoke `GenerateReportForOrder` with the `orderReportFormattingSettings` variable so that the signatures match.

The `HonestlyInject` method does not have such a requirement. It works even if the injected function has extra parameters. It moves such parameters to the resultant function. Therefore, if you look at the `generateReportForCustomer` variable, you will see that it has the following type:

```
Func<
    (
    Func<Customer, ImmutableArray<Order>>,
    Customer,
    OrderReportFormattingSettings
    ),
    CustomerReport>
```

Listing 12: The result of honestly injecting generateReportForOrder into ReportingModule.GenerateReportForCustomer

Notice how the resultant function has an `OrderReportFormattingSettings` parameter.

The same thing happened when we injected generateReportForCustomer into ReportingModule. `GenerateReportForCity`. The `Func<Customer, ImmutableArray<Order>>` parameter of `generateReportForCustomer` moved to the result of the injection, that is, the function stored in the `generateReportForCity` variable.

Figure 7 and 8 represents the Composition Roots (the Main methods) for the ReportGenerator7 project and the ReportGenerator8 project:

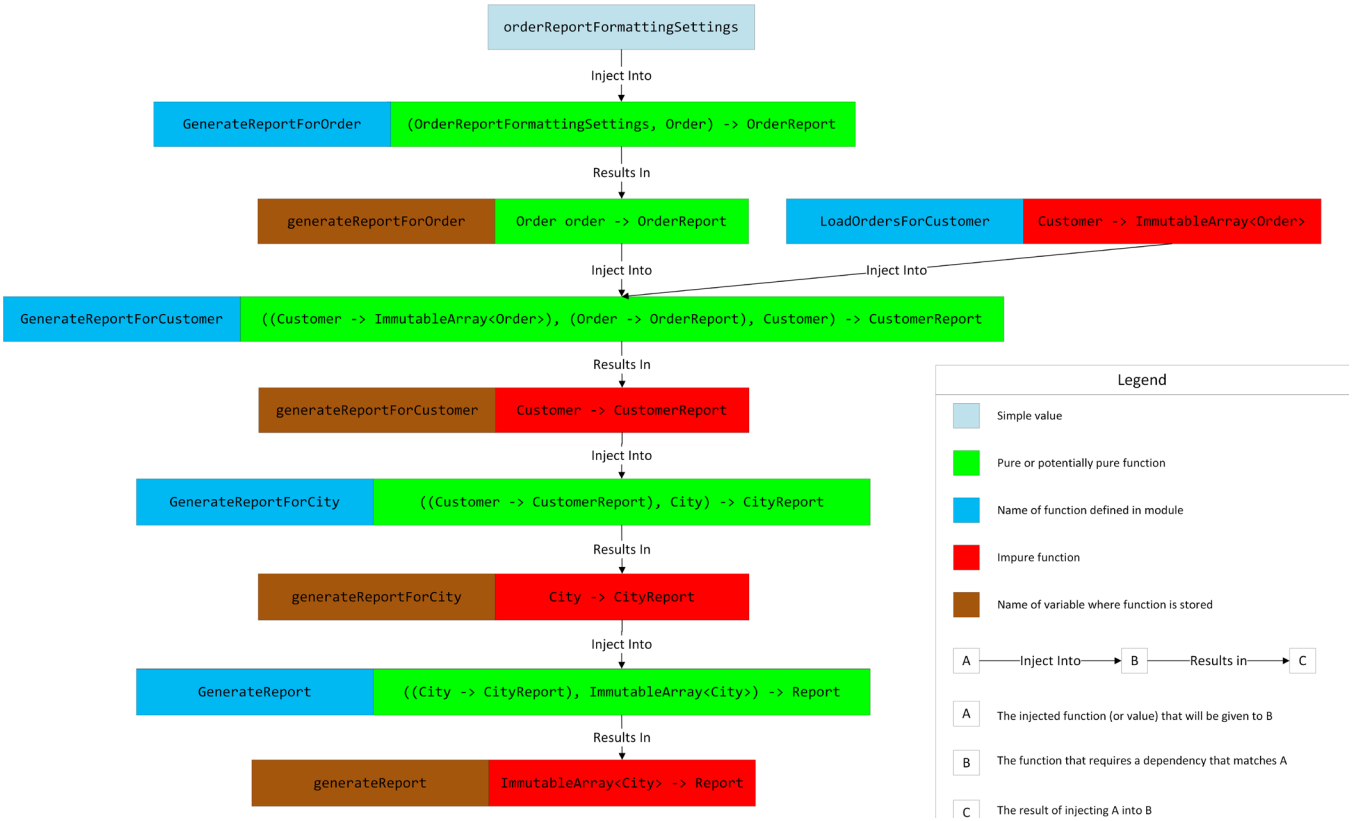


Figure 7: The Composition Root of ReportGenerator7

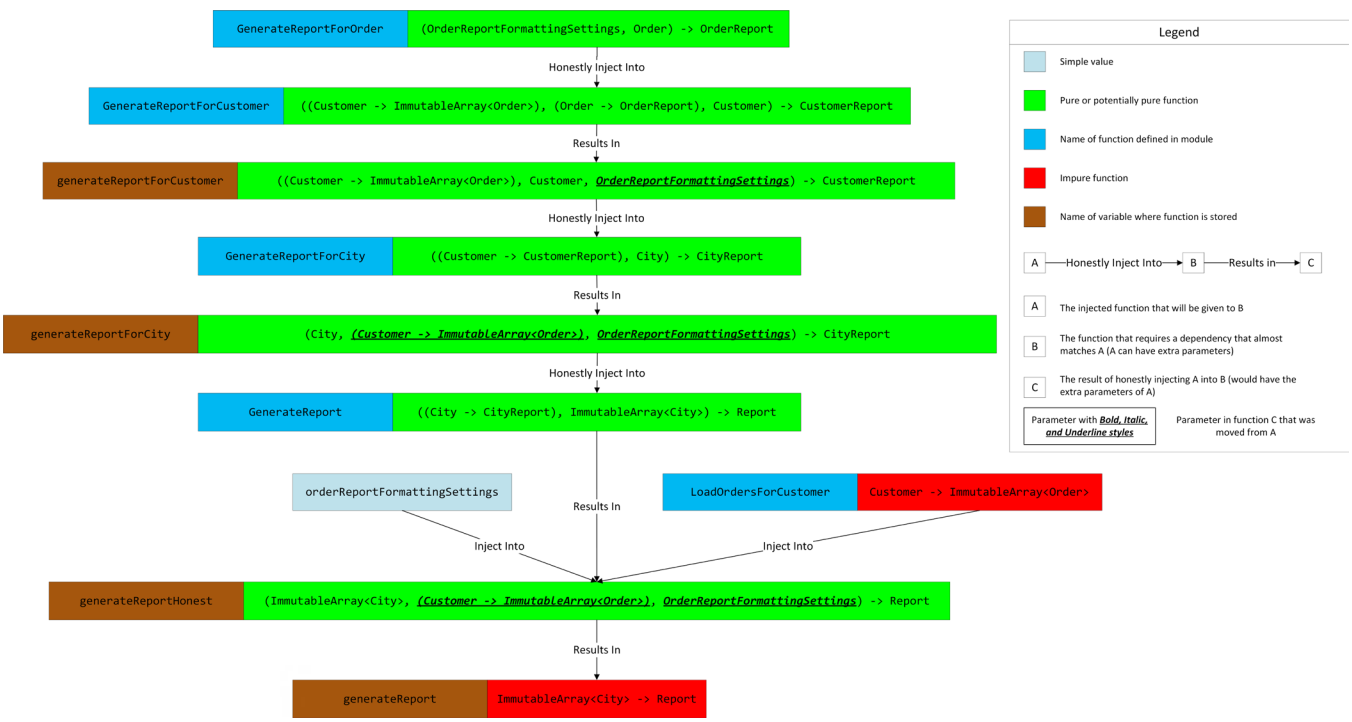


Figure 8: The Composition Root of ReportGenerator8

In summary, the `HonestlyInject` method allows us to delay the injection of impure functions to the last possible moment. This allows us to keep composing pure/potentially-pure functions to the last possible moment.

It is interesting to compare the composed functions in the `Main` method (in the ReportGenerator8 project) with the Generate* methods in Program.cs in the ReportGenerator4 project. Each composed function in ReportGenerator8 has the same parameters as the corresponding Generate* method in ReportGenerator4.

In some sense, ReportGenerator8 is a more maintainable version of the ReportGenerator4 project. I say "In some sense" because the technique I use in ReportGenerator8 has a lot of issues, see the last section of the article for more details.

The PurityAnalyzer extension can help us verify that the composed function, generateReportHonest, is in fact potentially-pure.

The Main method is impure. It will always be impure because at the end we need to inject impure dependencies.

Can PurityAnalyzer tell us that a part of the Main method is pure?

I have added a new feature in PurityAnalyzer 0.7 that allows developers to verify that some lambda expression is pure.

In PurityAnalyzer's settings (Tools > Options > Purity Analyzer), make sure you have the "Pure Lambda Full Class Name" and the "Pure Lambda Method Name" set like this:

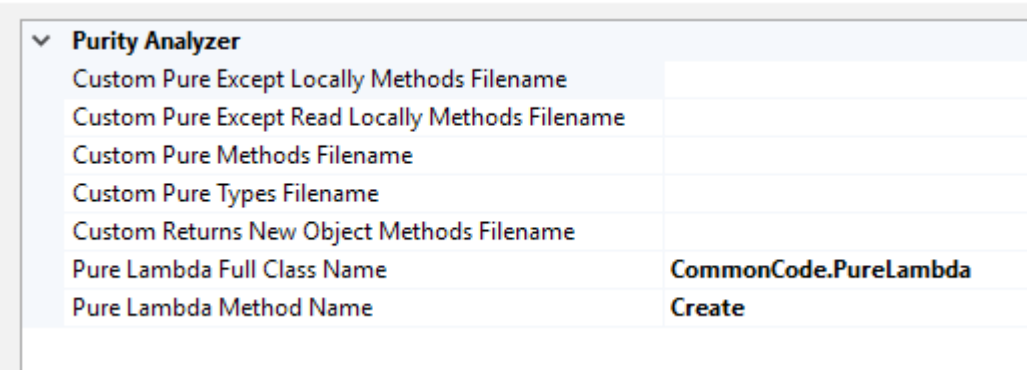


Figure 9: PureLambda settings

The `PureLambda.Create` method exists in the CommonCode project. This method takes a `Func<T>` parameter as input and returns the same `Func<T>` argument. When PurityAnalyzer sees this method called in code, it checks the purity of the lambda given to the method. This allows us to check the purity of a part of a method.

See the Main method in the ReportGenerator9 project to see how it is used. Try to inject (via `PartiallyInvoke`) the impure `DatabaseModule.LoadOrdersForCustomer` function into the composed function inside the lambda passed to `PureLambda.Create` and see how `PurityAnalyzer` objects.

The utility of the var keyword

I have used the `var` keyword to define all the intermediate function variables in the `Main` method.
This is required because if we change some low-level function to require some new dependency, and then go to the `Main` method and save it (to force the MappingCodeGenerator custom tool to run), the types of the variables holding the intermediate functions will change. If we use explicit types here, then we will have the same maintenance issue we had originally.

A New Technique

There are many programming languages that treat functions as a first-class citizen and that have the concept of partial invocation built into the language. However, as far as I know, there is not a language/framework that has the concept of honest dependency injection implemented into it.

I have been working on implementing a C# proof-of-concept for this for months now. I am using the extensibility features provided by Visual Studio and the features provided by the .NET Compiler Platform (Roslyn) to do so. These features of Visual Studio and the C# compiler are great in that they allow us to have something very close to adding new features to the C# language. Having said that, I think it would be great to have support for the concept of honest dependency injection in the C# language itself.

The following are some issues I see in the current solution that I have:

1. The names of the parameters are not maintained when calling `PartiallyInvoke` or when calling `HonestlyInject`. Currently, the parameters need to be distinguishable and understandable using their types alone. Even if a parameter is distinguishable using its type at the scope of the function itself, is it also distinguishable when the function is composed with many other functions?

2. How to deal with composite dependencies? For example, what to do when one function has a dependency on an array of functions? How to do honest dependency injection in this case?

3. If many low-level functions have a dependency on some `Func<string, Unit>` function used to log to the console, then the final generated function will contain many parameters of this type? How to deal with this?

4. What about performance? The generated code adds a lot of code layers. How does this affect performance?

5. The generated code makes navigating the non-generated code a bit more difficult.

6. What happens when the Composition Root becomes large? Can we split it into multiple methods? The return types of methods in C# cannot be 'var'. How does this affect maintainability?

7. Function parameters currently need to be in the format of `Func<(TInput1, TInput2, ..), TOutput>` to work with the extension correctly. For convenience, it should be possible to have function parameters with the type `Func<TInput1, TInput2,…, TOutput>`.

I will be talking about these issues and their potential solutions in the upcoming articles.

Conclusion:

In this article, I discussed the problem of making all methods/functions in an application pure or potentially-pure and provided a proof-of-concept for a solution.

When we make all the methods pure or potentially-pure, all the invocations of impure methods will be converted to function parameters passed into the potentially-pure methods. This means that all methods that end up calling the potentially pure methods will themselves need to have the function parameters required by the underlying methods. This will make the method signatures too long and will make adding a

new impure dependency for the low-level methods, a maintainability issue.

The solution is Dependency Injection, or partial invocation if we use functional programming terms.

However, the functions composed in the Composition Root become impure very quickly due to Dependency Injection. In some sense, this makes functions in the Composition Root, dishonest.

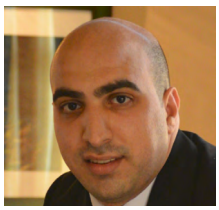
To fix this, I talked about a new technique which I call *Honest Dependency Injection*. Using this technique, we can inject pure/potentially-pure functions into each other without the need to inject impure dependencies first. This technique allows us to delay the injection of impure dependencies to the last possible moment.

• • • • • • •

Yacoub Massad

Author

Yacoub Massad is a software developer who works mainly with Microsoft technologies. Currently, he works at Zeva International where he uses C#, .NET, and other technologies to create eDiscovery solutions. He is interested in learning and writing about software design principles that aim at creating maintainable software. You can view his blog posts at criticalsoftwareblog.com.



Thanks to Damir Arh *for reviewing this article.*

DocuVieware 3

HTML5 Viewer & Document Management Kit

NEW RELEASE

Easy integration

Full support for custom snap-in

Zero-footprint solution

Fully customizable UI

Mobile devices optimization

Fast & crystal-clear rendering

Check the **New Features** and the **Online Demos**

DOWNLOAD
YOUR FREE TRIAL

www.docuvieware.com

Powered by GdPicture.NET 14

DocuVieware is an ORPALIS imaging technologies product

Ravi Kiran

ReactJS Router Basics

Routing adds usefulness and richness to a front-end application. This article will get you started with adding such richness to React.js applications using the React Router.

Routing is an important part of every front-end application.

It allows to organize the application as small chunks. At the same time, it provides a user-friendly experience as users can switch between different parts of the application, bookmark URLs, share URLs with others and use browser's history to navigate between the states of the application that the user has seen before.

A ReactJS application is no different. This article will explain you the process of adding routes to a react application.

Note: *This article assumes that you have a basic understanding of React and know how it works. If you are not yet familiar with React, you may check www.dotnetcurry.com/reactjs/1353/react-js-tutorial to get started.*

ReactJS Application Setup

To follow along and create the demo application, install the following software:

- Node.js
- VS Code
- npm 5.2 or above
- yarn

After installing Node.js, check the version of npm and if it is below 5.2, you can upgrade using the following command:

```
> npm update -g npm
```

The reason for upgrading *npm* is from version 5.2, npm comes with the package runner tool npx.

npx helps in reducing time and energy while using global npm packages. To create a ReactJS application, we need the package *create-react-app*. With npx, we can use the create-react-app package without installing it globally. The following npx command creates the new application:

```
> npx create-react-app my-app
```

Once the application is created and the packages are installed, it can be executed using one of the following commands:

```
> npm start
```
or
```
> yarn start
```

This command starts the React application on port 3000.

Building a ReactJS Application

Let's build an application to explore features of React router. The application will use the GitHub API to fetch details of a GitHub organization. Then it will display the list of repos and the list of members on different pages. The application will have one more page to show the details of a member, this page will be reachable through the members list page.

First, let's create the application and install the required packages in it. Run the following command to generate the project:

```
> npx create-react-app github-org
```

Move to the newly created folder on a terminal. Run the application using the *npm* start command and you will see the basic react application running on the browser.

Let's install the packages required.

We need to install *react-router-dom* package to use routes and *bootstrap* to use styles in the application. The following commands install these packages:

```
> yarn add react-router-dom@4.3.1
> yarn add bootstrap
```

Now these packages can be imported into the application and used. The following statement imports Bootstrap's CSS file into the application:

```
import 'bootstrap/dist/css/bootstrap.css';
```

Let's add some routes to the application.

To use the routes, the *App* component has to be rendered inside the *BrowserRouter* component. This component adds the capabilities of routing to the React web application. Open the file *index.js* and change the code in this file as shown here:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter } from 'react-router-dom';
import './index.css';
import App from './App';
import registerServiceWorker from './registerServiceWorker';

ReactDOM.render((
<BrowserRouter>
  <App />
</BrowserRouter>
), document.getElementById('root'));
registerServiceWorker();
```

Adding Routes to Show Repos and Members

Now we can define routes and use them. Before creating a route, let's create a component to be rendered inside the route. Add a new file to the *src* folder and name it *Repos.js*. Add the following code to this file:

```
import React, { Component } from 'react';

export class Repos extends Component {
  render() {
    return <div>We will list the repos here!</div>;
  }
}
```

This component can be used to define a route. Open the file *App.js* and change the code in this file as shown here:

```
import React, { Component } from 'react';
import { Route } from 'react-router-dom';  // 1
import logo from './logo.svg';
import 'bootstrap/dist/css/bootstrap.css';
import './App.css';
import { Repos } from './Repos';  // 2

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
```

```
          <h1 className="App-title">Welcome to React</h1>
        </header>
        <Route path="/repos" component={Repos}></Route>  {/* 3 */}
      </div>
    );
  }
}

export default App;
```

The new or modified lines in the above snippet are marked with comments and are explained as follows:\

1. Importing the *Route* component
2. Importing the new component to be rendered inside the route
3. This statement defines the route. When React Router encounters the relative path /repos, the component *Repo* is rendered in place of the Route component. The *Route* component performs two things here. One, it configures the route and tells what to do when a URL is reached. The other, it acts as the placeholder for the component referred by the route.

Save these changes and run the application. You will see only the header alone rendered on the page. This is because, the browser has the URL http://localhost:3000 and it points to the root of the application. Change the URL to http://localhost:3000/repos and you will see the content of the new component on the page.

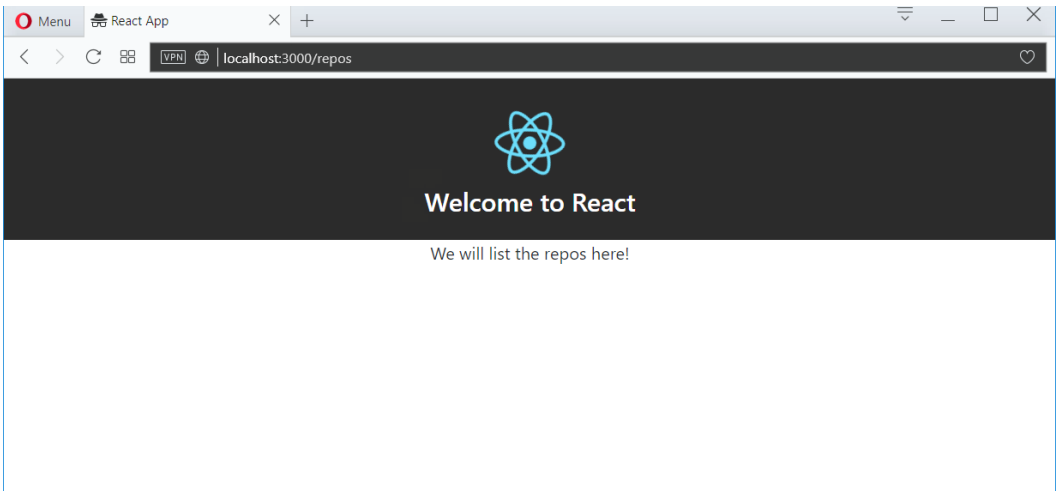


Figure 1: React application with one route

To render this component by default on the page, the route configuration has to be modified. Change the Route component in the file *App.js* as follows:

```
<Route path="/" component={Repos}></Route>
```

Change the URL to http://localhost:3000 after saving this change and now you will see the component *Repos* rendered.

Let's modify the *Repos* component to show the repositories under the Facebook organization on GitHub. To separate the logic of data fetching, the code interacting with the GitHub API will be kept in a separate class. Add a new file and name it *githubData.js*. Add the following code to this file:

```
class GithubData {
 org = 'facebook';
 orgBaseUrl = 'https://api.github.com/orgs/';
```

```
  constructor() {
    this.headers = new Headers();
    this.headers.append('Accept', 'application/vnd.github.v3+json');
  }

  getRepos(orgName) {
    let request = new Request(`${this.orgBaseUrl}${this.org}/repos`, { headers:
this.headers });

    return fetch(request).then((res) => {
      return res.json();
    });
  }
}

export let githubDataProvider = new GithubData();
```

This snippet uses the Fetch API to call the GitHub API. The *getRepos* method gets the list of repositories and returns it. While sending a request to the GitHub API, it is mandatory to send the header parameter *Accept*. So, add this setting to the property *headers* in the class, so that it can be reused.

Now we need to modify the *Repos* component to use this data. The component will display this data in a table. The structure of the table would be created in the *render* method and the data would be fetched in the *componentDidMount* method, once the component is ready on the page.

The *render* method will return the JSX that is initially needed to show on the page. The *componentDidMount* method will build JSX for the content of the table once the data is fetched. The following snippet shows code of the modified component:

```
import React, { Component } from "react";
import { githubDataSvc } from "./githubData";

export class Repos extends Component {
 constructor() {
  super();
  this.state = {
   repos: []
  };
 }
 componentDidMount() {
  githubDataSvc
   .getRepos()
   .then(repos => this.setState({ repos }));
 }
 renderRepoInfo(repo) {
  const link = repo.homepage ? (<a href={repo.homepage} target="_blank">Home Page</
  a>) : (<span>-</span>);
  return (
   <tr key={repo.id}>
    <td> {repo.name} </td>
    <td> {repo.stargazers_count} </td>
    <td> {repo.watchers_count} </td>
    <td> {repo.forks_count} </td>
    <td> {repo.open_issues_count} </td>
    <td>{link}</td>
   </tr>
  );
 }
```

```
 render() {
  return (
   <div>
    <h3>Repos in the org {githubDataSvc.org}</h3>
     <table className=”table”>
      <thead>
       <tr>
        <th>Name</th>
        <th>Stars</th>
        <th>Watches</th>
        <th>Forks</th>
        <th>issues</th>
        <th>Home Page</th>
       </tr>
      </thead>
      <tbody>{this.state.repos.map(this.renderRepoInfo)}</tbody>
     </table>
    </div>
   );
 }
}
```

As you see, body of the table is bound to the *repos* property on the state of the component. This property is initialized with an empty array and the array is filled with data after the data is loaded from the GitHub API. Once the page is rendered, you will see a table similar to Figure 2.

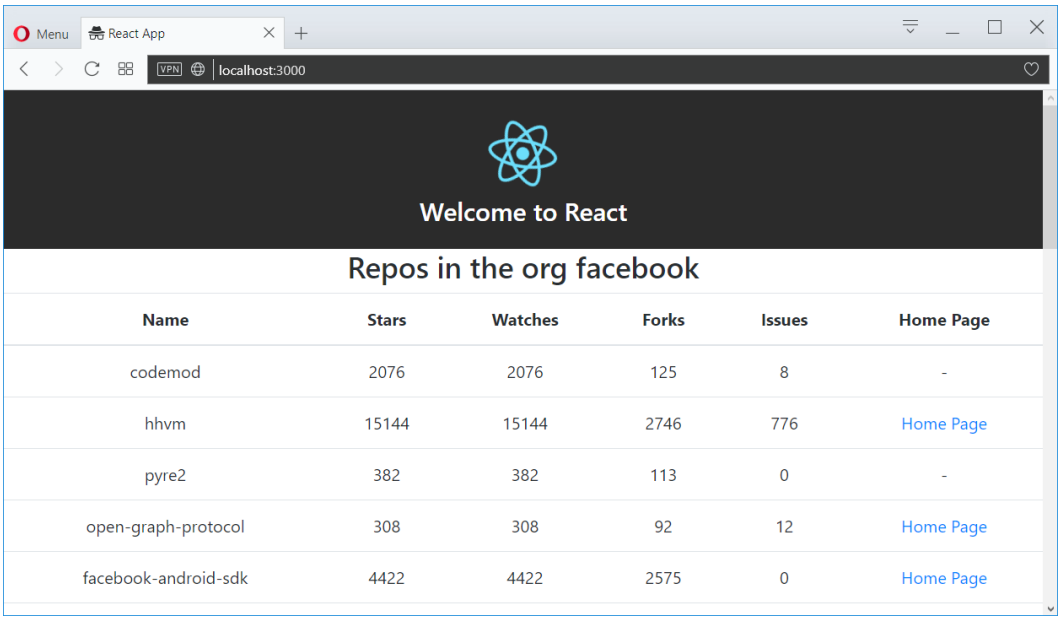


Figure 2: Home page with the repos in Facebook org

Now that the repos page is ready, let's add another page to show the members in the organization. To fetch the members of the organization, add the following code to the *GithubData* class:

```
getMembers(orgName) {
  let request =  new Request(`${this.orgBaseUrl}${this.org}/members`, { headers:
  this.headers });
  return fetch(request).then((res) => {
    return res.json()
      .then(members => {
        return members;
    });
  });
}
```

Add a new file to the *src* folder and name it *Members.js*. The *Members* component will have a similar structure as the Repos component. It will build the initial content in the *render* method and the members list will be created in the *componentDidMount* method.

Instead of showing the list of members in a table, it will create a tile view to display the avatars of the members and will have a few links to the profile of the member. The following snippet shows code of this component:

```
import React, { Component } from “react”;
import { githubDataSvc } from “./githubData”;
import { Link } from “react-router-dom”;

export class Members extends Component {
 constructor() {
  super();
  this.state = {
   members: []
  };
 }

 componentDidMount() {
  githubDataSvc
   .getMembers()
   .then(members => this.setState({ members }));
 }

 renderMember(member) {
  return (
   <div className=”col-md-3 card” key={member.login}>
    <div className=”card-body”>
     <div>
      <img
       src={member.avatar_url}
       alt={member.login}
       className=”avatar”
      />
     </div>
     <div>
      <a target=”_blank” className=”text-left” href={member.html_url}>
       {member.login}
      </a>
      <br />
      <a target=”_blank” href={member.html_url + “?tab=repositories”}>
       Repos
      </a>
      <br />
      <Link to={`/member/${member.login}`}>View Details</Link>
     </div>
    </div>
   </div>
  );
}

render() {
 return (
  <div>
   <h3>Members of the {githubDataSvc.org} organization</h3>
    <div className=”row”>
     {this.state.members.map(this.renderMember)}
    </div>
```

```
    </div>
   );
  }
}
```

The *Members* component uses the following styles to format the images and the tiles, add them to the *App.css* file:

```
.avatar{
  height: 170px;
  width: 170px;
}

.tile{
  margin-top: 10px;
}
```

Now the *App* component has to be modified to show either *Repos* component or the *Members* component depending on the URL. The *Members* component will have its own *Route* configuration. To show one of these routes, we need to use the *Switch* component. Change the code in the file *App.js* as shown here:

```
import React, { Component } from 'react';
import { Route, Switch } from 'react-router-dom'; // 1
import logo from './logo.svg';
import 'bootstrap/dist/css/bootstrap.css';
import './App.css';
import { Repos } from './Repos'; // 2
import { Members } from './Members';

class App extends Component {
 render() {
  return (
   <div className="App">
    <header className="App-header">
     <img src={logo} className="App-logo" alt="logo" />
     <h1 className="App-title">Welcome to React</h1>
    </header>
    <Switch>
     <Route path='/' component={Repos}></Route>
     <Route path='/members' component={Members}></Route>
    </Switch>
   </div>
  );
 }
}

export default App;
```

Now run the application and change the URL to http://localhost:3000/members. You will see that the page still shows the *Repos* component. This is because, the relative path /members also matches the path /.

To fix this, add the attribute *exact* to the route. The modified route is shown here:

```
<Route path='/' exact component={Repos}>
```

Now you will see the *Members* component on the page when the URL is changed to http://localhost:3000/members.

Adding Links

It would be great to have links to switch between these views. Links to the react routes can be generated using the *Link* component of React router. It can be imported in the same way as the *Route* component and can be used as shown below:

```
<Link to='/'>Repos</Link>
```

Here the prop ***to*** accepts the target path. The links will be added to a left menu in the file *App.js*. To show the menu, view of the file *App.js* has to be changed to show the menu and the content. Change contents of the file *App.js* as shown here:

```
import React, { Component } from 'react';
import { Link, Route, Switch } from 'react-router-dom';
import logo from './logo.svg';
import 'bootstrap/dist/css/bootstrap.css';
import './App.css';
import { Repos } from './Repos';
import { Members } from './Members';

class App extends Component {
 render() {
  return (
   <div className="App">
    <header className="App-header">
     <img src={logo} className="App-logo" alt="logo" />
      <h1 className="App-title">Explore the Facebook org on GitHub</h1>
    </header>
    <section className="content">
     <div className="container row">
      <div className="col-md-3">
       <nav className="navbar">
        <ul className="navbar-nav">
        <li className="nav-item">
         <Link className="nav-link" to='/' >Repos</Link>
        </li>
        <li className="nav-item">
         <Link className="nav-link" to='/members'>Members</Link>
        </li>
       </ul>
      </nav>
     </div>
     <div className="col-md-9">
      <Switch>
       <Route path="/" exact component={Repos} />
       <Route path="/members" component={Members} />
      </Switch>
     </div>
    </div>
   </section>
  </div>
  );
 }
}

export default App;
```

Now you will see the menu and the content of the route rendered. You can switch between the two routes using the links. Figure 3 shows how the page looks now:

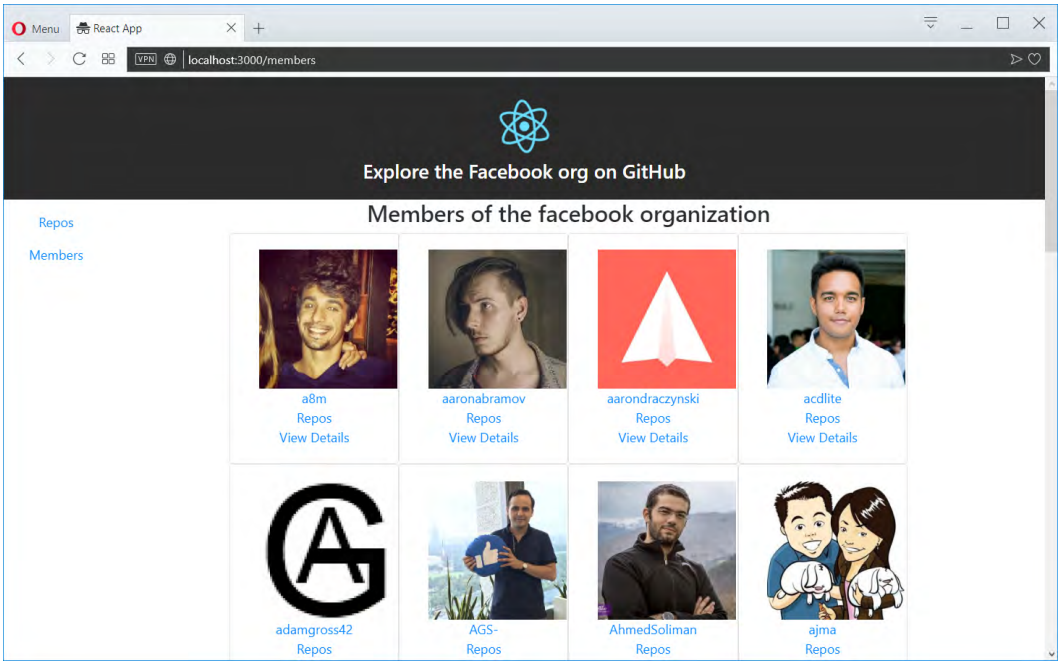


Figure 3: Members page

Passing Parameter

It would be good to create a page to show the details of a member. This page can be accessed through the member list page and will display some basic details of the member along with the list of repos the member has in her/his GitHub account.

The members list page will have links to the details pages of the corresponding members and the links will pass login id of the member to the details page. The details page would use the login id to query the GitHub API to get the details of the member and the list of repos the member has.

Add a new file to the *src* folder and name it *MemberDetails.js*. Add the following code to it:

```
import React, { Component } from 'react';
import { githubDataProvider } from './githubData';

export class MemberDetails extends Component {
  constructor(props) {
    super(props);
  }
  render() {
    return <div>We will see details of the user {this.props.match.params.login}
    here shortly...</div>;
  }
}
```

The parameter passed to the route would be made available to the component as part of its *props*. Notice the *div* element returned from the *render* method, it binds the parameter it accepted using *this.props.match.params.login*.

Here name of the parameter is *login*. The same name has to be used while configuring the route. Let's define the route to load this component now. Open the file *App.js* and add the following import statement:

```
import { MemberDetails } from './MemberDetails';
```

Modify the *Switch* component as shown here:

```
<Switch>
  <Route  path=”/ “  component={Repos}  />
  <Route  path=”/members”  component={Members}  />
  <Route  path=”/member/:login”  component={MemberDetails}  />
</Switch>
```

Now if you run the application and change the URL to http://localhost:3000/member/aaronabramov, you will see the following page as shown in Figure 4:

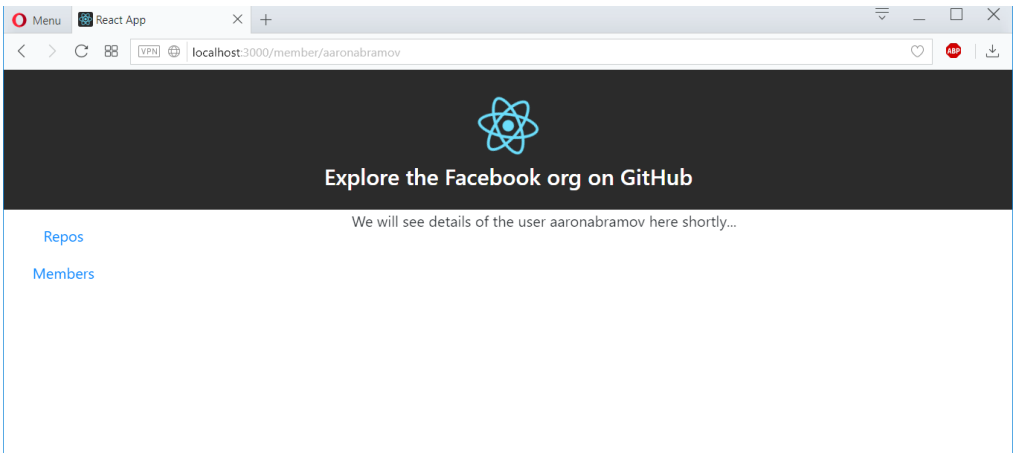


Figure 4: Route with parameter

To add links to the details pages from the members list page, we need to import the *Link* component and pass the login id of the user. Open the file *Members.js* and add the following JSX after the two anchor tags in the *membersjsx* variable:

```
<br  />
<Link  to={`/member/${member.login}`}>View Details</Link>
```

On running the application now, you will see the links to the details page rendered in the members page and on clicking any of these links, you will reach the member details page.

Now that we are able to get the login of the user in the member details page, let’s use it to fetch the required data and bind it. Open the file *githubData.js* file and add the following method in the class:

```
getMemberDetails(memberLogin) {
  let userRequest = new Request(`${this.userBaseUrl}${memberLogin}`, { headers:
this.headers });
  let userReposRequest = new Request(`${this.userBaseUrl}${memberLogin}/repos`, {
  headers: this.headers });

  return Promise.all([fetch(userRequest),
  fetch(userReposRequest)])
    .then(([member, repos]) => {
      return Promise.all([member.json(), repos.json()]);
    })
    .then(([memberValue, reposValue]) => {
      return {
        member: memberValue,
        repos: reposValue
    };
  });
```

The last thing to do is, make the *MemberDetails.js* file functional. Open this file and replace the code in this file with the following:

```
import React, { Component } from “react”;
import { githubDataSvc } from “./githubData”;

export class MemberDetails extends Component {
 constructor(props) {
  super(props);
   this.state = {
    member: null,
    repos: []
   };
 }

 get Login() {
  return this.props.match.params.login;
 }

 componentDidMount() {
  githubDataSvc
   .getMemberDetails(this.Login)
   .then(({ member, repos }) => this.setState({ member, repos }));
 }

 renderMemberDetails() {
  const { member, repos } = this.state;

  if (!member) {
   return null;
  }

  return (
   <div className=”row”>
   <div className=”col-md-6”>
   <div className=”row”>
   <div className=”col-md-6”>Name: </div>
   <div className=”col-md-6”>{member.name}</div>
   <div className=”col-md-6”>Company: </div>
   <div className=”col-md-6”>{member.company}</div>
   <div className=”col-md-6”>Number of Repos: </div>
   <div className=”col-md-6”>{member.public_repos}</div>
   <div className=”col-md-6”>Number of Followers: </div>
   <div className=”col-md-6”>{member.followers}</div>
   <div className=”col-md-6”>Following Count: </div>
   <div className=”col-md-6”>{member.following}</div>
  </div>
 </div>
 <div className=”col-md-6”>
  <img
   src={member.avatar_url}
   alt={`${member.name}’s pic`}
   height=”150”
   width=”150”
  />
 </div>

 <h4>{member.name}’s Repos</h4>

 <table className=”table”>
  <thead>
```

```
    <tr>
     <th>Name</th>
     <th>Stars</th>
     <th>Watches</th>
     <th>Forks</th>
     <th>issues</th>
     <th>Home Page</th>
    </tr>
   </thead>
   <tbody>{repos.map(this.renderRepo)}</tbody>
  </table>
</div>
);
}

  renderRepo(repo) {
   let link = repo.homepage ? (<a href={repo.homepage} target="_blank">Home Page</
   a>) : (<span>-</span>);

   return (
    <tr key={repo.id}>
     <td> {repo.name} </td>
     <td> {repo.stargazers_count} </td>
     <td> {repo.watchers_count} </td>
     <td> {repo.forks_count} </td>
     <td> {repo.open_issues_count} </td>
     <td>{link}</td>
    </tr>
   );
  }

  render() {
   return (<div>{this.renderMemberDetails()}</div>);
  }
}
```

Now you will see the following page in the browser as shown in Figure 5 when you visit the URL http://localhost:3000/member/aaronabramov:

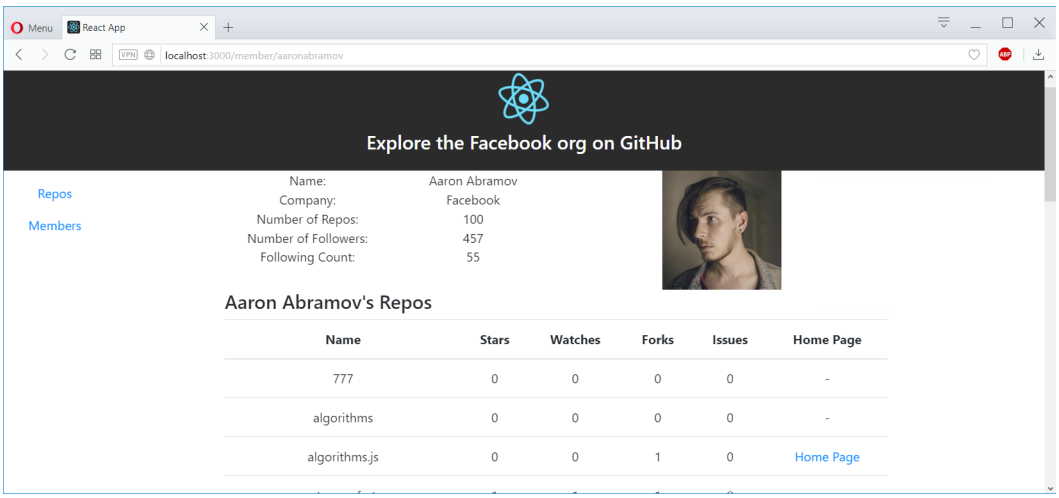


Figure 5: Member details page

ReactJS Router vs Angular Router

If you are familiar with Angular before learning React, you will naturally compare the two frameworks. If you have worked with Angular's router (Check Keerti's Article on Page 74, you will clearly notice the differences between the routers of the two frameworks.

Following is a list that compares the routers:

- Angular's router is built into the framework and is shipped as part of the framework, while React's most popular router is built by a group of open source developers outside the official team at Facebook
- Routes in an Angular application are configured in a module and this module is imported into the module rendering routes. React's routes are configured as part of a component's view
- Both Angular and ReactJS support dynamic route configuration. Meaning, configuration of the routes can be changed based on some condition in both the frameworks
- Both the routers are based on components and they render the component when the corresponding path is encountered

Editorial Note: *You may also want to read about the difference between Angular vs React vs Vue at www.dotnetcurry.com/vuejs/1372/vuejs-vs-angular-reactjs-compare*

Conclusion

React router DOM provides a simple yet powerful way to create routes and use them in browser based React applications. As we saw, routes can be added to React applications very quickly. We will explore more features of the router in a forthcoming article.

Download the entire source code from GitHub at
bit.ly/dncm39-reactjs-routing

Microsoft® Most Valuable Professional

Ravi Kiran
Author

Ravi Kiran (a.k.a. Ravi Kiran) is a developer working on Microsoft Technologies at Hyderabad. These days, he is spending his time on JavaScript frameworks like AngularJS, latest updates to JavaScript in ES6 and ES7, Web Components, Node.js and also on several Microsoft technologies including ASP.NET 5, SignalR and C#. He is an active blogger, an author at SitePoint and at DotNetCurry. He is rewarded with Microsoft MVP (ASP.NET/IIS) and DZone MVB awards for his contribution to the community.

Thanks to Shravan Kumar Kasagoni for reviewing this article.

COVERS C# 7, C# 8 AND .NET Core

THE ABSOLUTELY AWESOME

BOOK ON

PREORDER NOW!

Rahul Sahasrabuddhe

A Blueprint for Building a Center Of Excellence

Have you heard about the swanky/ cool group or team names in your organizations like Center of Excellence (CoE) for SAP, Microsoft Competency Center or SFDC (i.e. Salesforce) Technology Practice and so on? Have these terms made you wonder what they are and what purpose do they serve, besides appearing cool.

This article is an attempt to provide you a perspective about these groups, and what they do, by providing a generic blueprint for building a CoE.

Introduction: What's a CoE, anyway?

A CoE (and we will use this all-encompassing term going forward for all such initiatives) is something that is built with certain business and/or technology goals in mind.

Organizations often use the strategy of focusing on few specific technology areas and banking on them for growth by aligning all their resources (man-power, investments etc.) towards it. Focusing on one or a few things always works out better than doing too many things at a time and not getting a return on investment (ROI) on any of them.

Usually, the high-level goals of building a CoE are:

1. Turning vision into action
2. Riding the hype cycle of a specific technology area or a vertical to grow business
3. Using CoE as a vehicle to take the organization to the next level of scale in terms of revenue, market share, etc.
4. Edge the competition out by being better/superior to them in a specific technology area

Before we move on, such CoEs are conceptualized in organizations in the following two scenarios:

1. Organizations that offer product engineering services or product development services set up such CoEs. It is because they want to diversify their offerings for technology stacks and be in line with the changing technology landscape to aid the business growth. Product companies may also setup such CoEs to focus on specific technologies that their products are using.

2. Service providers and system integrators (SIs) set up CoEs in organizations (or Enterprises) for consulting and training on that service or platform (for example, SDFC or SAP). The purpose is to provide a comprehensive guidance regarding use of a specific technology, create organization wide standards regarding these technologies and to spearhead the development in those. These are the people who do not work for a single project or product but provide internal consulting to all teams.

In this article, I present a generic methodology that can be used for building a CoE.

As always, this blueprint is based on my endeavors (or adventures) and experiences of managing and building CoEs. So, this blueprint, in no way, is THE blueprint to be followed. You are smart enough to come up with your own, as long as you know you will succeed!

What's your CoE?

CoEs could be built for any of the following:

1. **Technology or platform:** You could focus on a specific upcoming technology to build a team. For example, UI (JavaScript) technologies could be an area that your organization would build a CoE for due to a heavy demand for that skill (and hence business growth).

2. **A specific vertical or a domain:** An organization may want to focus on healthcare as a vertical and HMS (hospital management system) as a specific sub-vertical. Or the key leaders in the organization (you could be one of them) think that there is a good buzz about IIoT (or industrial IoT) as a sub-vertical or area and hence they may want to build a CoE for that.

3. **A specific software or product offering:** There are multiple kinds of product types available for serving a specific customer need. For example, CRM, ERP, CMS and so on. So, an organization may want to build expertise in such specific software viz. CRM.

Do you have to be a CXO (CTO/CIO/CEO) of some company to build a CoE?

Well, yes and no.

If you are a CXO level person, then you have access to all the resources to make decisions. However, even if you are a project manager or a developer, you can still use this blueprint to build expertise at your individual level with respect to some technology area. The core principles would still hold true.

While building a CoE, you would apply a ***breadth first; depth later*** approach since you would want to gain an overall understanding of a technology first, and then go deeper in any specific area.

The CoE Blueprint

Following is the blueprint for building a CoE. It has 4 phases as shown below.

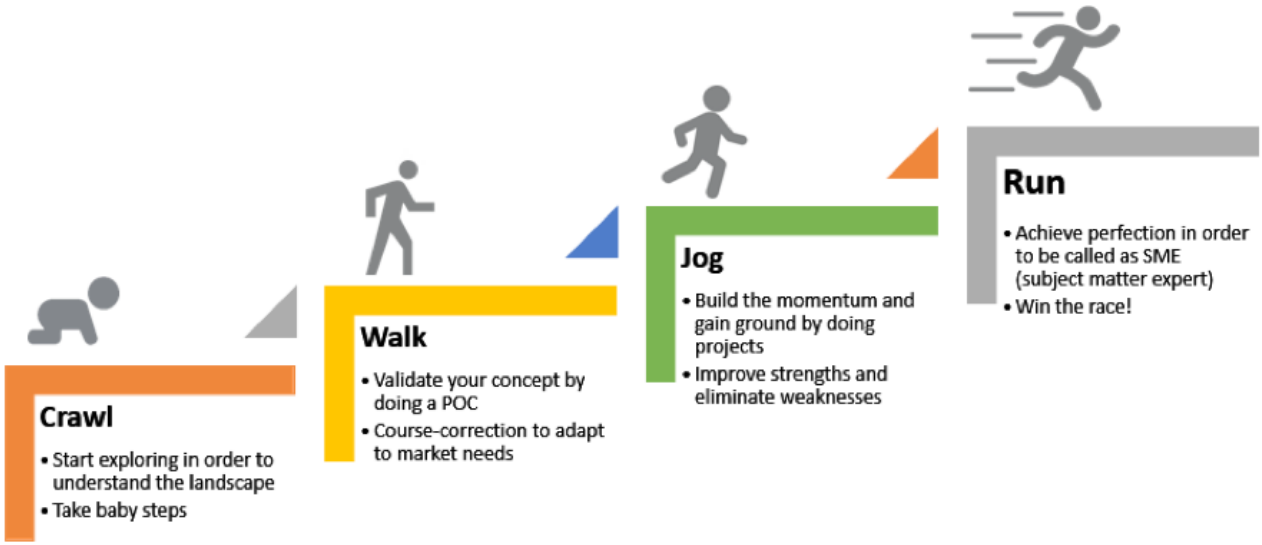


Figure 1 : The CoE Blueprint

The analogy used here is how you start crawling in your childhood, to how you eventually grow in strength and confidence to run fast and win races (not actually, but analogous to achieving your desired goals in life).

In the upcoming sections, for each phase, I will elaborate on:

- the pre-requisites for the phase
- various steps to be taken
- the timeframe needed, and
- the KRAs (the key result areas) to be fulfilled in each phase.

This looks cool, eh?

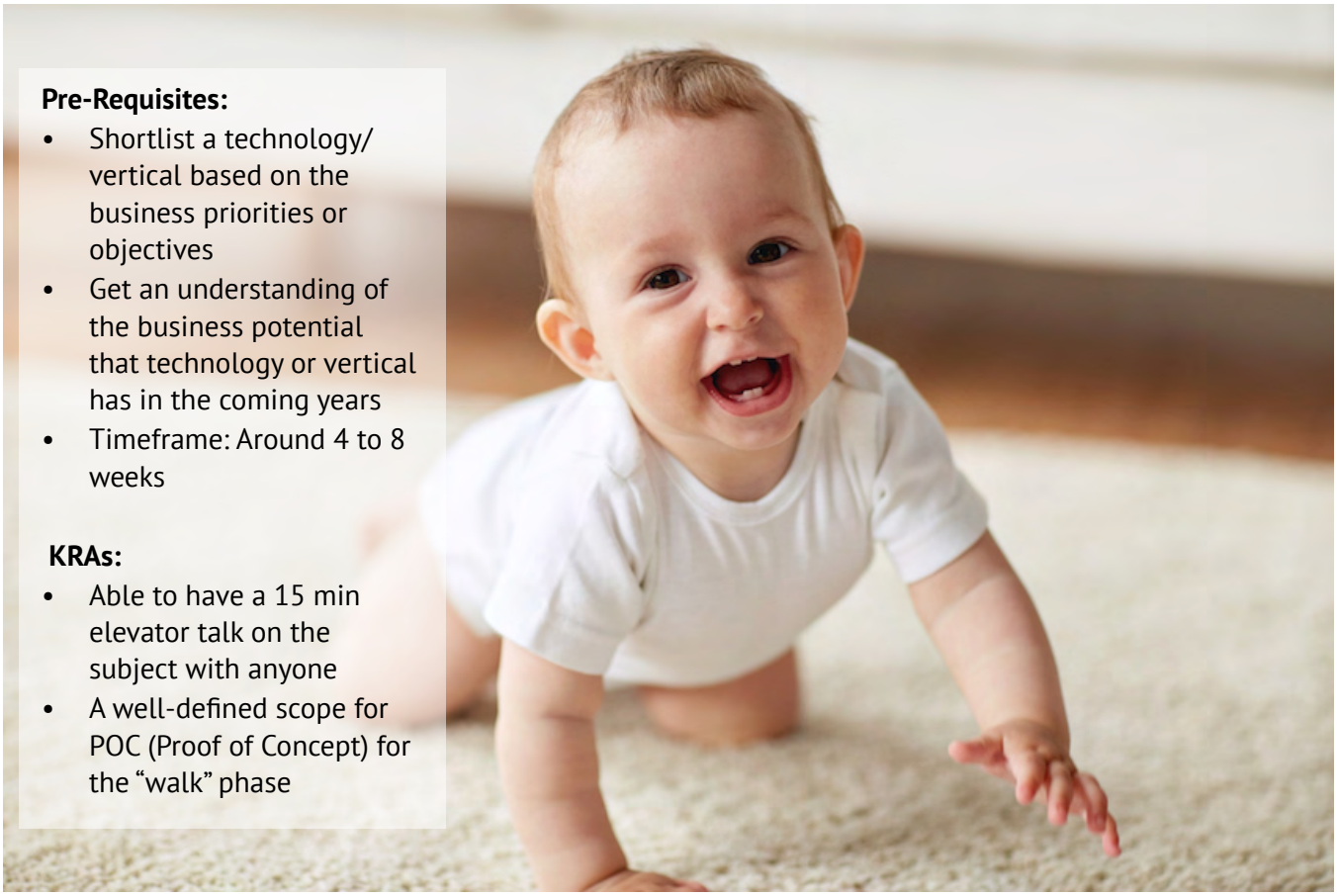
But the proof of the pudding is in the eating. So, while we discuss the blueprint, let us also apply this model and see how we can build a CoE for one of the hottest and happening technology areas – AI or artificial intelligence.

Let's call our CoE for AI as AI-CC.

In upcoming sections, at the end of each phase, I will mention about how that phase would pan out for AICC. This will give you a good idea of how this whole conceptual model works in reality. Please look for an AICC section for each phase.

Ready to do some exercise?

Phase 1: Crawl



Pre-requisites

Before embarking upon the journey of building a CoE, you need to have a high-level idea about a specific technology or vertical in which you would like to build a CoE.

Although it is important to know where you want to go, it is not necessary that you will wind up there. At this stage of building a CoE, you are allowed to go a few steps forward (or crawl for some distance) and then stop and move elsewhere (i.e. pursue other technology area).

Another important aspect is to get a buy-in from key stakeholders about you or your team wishing to spend time on exploring a specific technology area. This aspect will drive the decisions later to funding/ investment once the CoE takes some shape (especially the POC-phase).

To choose the technology area, you need to have a clarity about:

- Why is that area important to you, business growth-wise? Is it because everyone is building CoE in that area or it's because you are convinced that there is a future in it?

- Does that area have business potential and have you validated it? You can refer to the concepts of blue ocean vs. red ocean strategy to understand this aspect better here.

- As an individual, do you think you will benefit from learning about this area as it will add to versatility of your skills?

The simplest way to get a clarity on this is to read up about industry trends, upcoming tech/business areas and so on and form your opinion based on that.

If you are already an avid reader of technology, this should not be too difficult for you.

Crawling into the CoE

Following are the steps you would follow in the crawling process:

1. First and foremost, you do enough market research to know about various possible technology areas or verticals. You would end up reading technical magazines like DotNetCurry (DNC). You would even check out what your competitors (in business) or peers (at an individual level) are learning about.

2. After zeroing in on 3-4 areas, now is the time to read up or understand more about specific technology areas and compare and contrast.

3. Typically, the crawl phase would go on for 4 to 8 weeks.

Once you have done this, you can assess the effectiveness of this phase by referring to following KRAs.

KRAs

The key result areas for successful completion of this phase should be:

1. Have a reasonable understanding of the spectrum of technology or vertical in which you want to build the CoE.

2. You should be able to have a 10-15 minute elevator conversation with anyone about this technology or vertical after this phase is over. If you can have this impromptu conversation with anyone who understands technology and if that person thinks you are comfortable with a specific technology, then, my friend, you have cleared the 1st phase with distinction!

3. You should have some scenarios identified that can be a good starting point for doing a POC (i.e. Proof of Concept).

Following are some important considerations in finalizing the scope of a POC:

1. The scenarios shouldn't be too complicated in terms of functional or non-functional aspects as a POC to demonstrate these aspects would eat up a lot of time.

2. They should be relevant for customers/consumers of your organization so you can demonstrate the usefulness of that technology area well enough.

The AICC Crawl

In the context of AICC, we have already completed part of the crawl phase by choosing to build competency in AI. However, to validate the model, you will need to:

1. Get a good understanding about why and how AI is becoming so popular. You will find umpteen articles/ blogs about this aspect. Once you do, please take up the 15 min elevator conversation test and see how you perform.

2. Next up, you need to come up with scenarios for building a bot using an AI framework to address a specific business need or requirement. Here is a quick-start link for you.

Phase 2: Walk

Pre-requisites:

- Good understanding about a technology or vertical, chosen in the "Crawl" phase.
- Having right resources to build a POC (effort/cost and skills)
- Timeframe: 1 to 3 months

KRAs:

- A demonstrable POC that can highlight the understanding you have about the technology/area
- Completing certifications and enrolling in partner programs
- A business plan/strategy in place for achieving long term goals and an offering crafted that can be used for getting business in the "Jog" phase

Pre-requisites

Assuming that you have crawled up to here (not literally), you would now have a good understanding of the technology or vertical in which you are building a CoE. It's one of the key pre-requisites.

But that is not good enough!

What you need now is that the key stakeholders in your organization (basically your bosses) are aligned well with the CoE objectives. So, you need to get a buy-in from them. It is in this phase when you need them to sign off on important things like strategy to build the CoE, corresponding investments needed etc.

The key aspects that you would need to make this phase a success are:

1. Having people with right skills to build a POC or POCs for CoE. Usually you would hire some experts from the market and bank on their knowledge or experience. In addition to this, you identify smart people from within the organization and train them on respective technologies. The reason to follow this two-pronged strategy is that it keeps the costs down and it also helps in building the team from within. Initially you may want to start by building in-house expertise.

2. For specific technology areas or verticals, you would also need to spend money in having licenses for right software, infrastructure and so on. So, if you are going to build a SAP CoE, you would need to have some money allocated to acquire licenses of SAP.

While doing so, it is important to start small in terms of investments – be it people or tools/licenses. It helps to start with a small core seed team of people and then add more members once you start seeing traction.

When you are looking for people within the organization for building a CoE, you need open-minded creative people who don't mind doing something new every day and throwing it off the next day just because the priorities changed overnight.

You usually wouldn't want people who like doing fixed set of things every day. They are useful for your CoE when you have many projects coming in.

Start Walking

In this process, you will be doing the following:

1. Since you have a good understanding about what the technology can and cannot do and also what the market needs are, you would spend time in finalizing a business scenario (or scenarios) to implement a POC. The objectives of building a POC are:
   - The business use case chosen is simple enough to implement in a short period of time (remember, a POC cannot run for months).
   - POC should not take more than 3 to 4 weeks to implement.
   - You may not need to have experts to do the POC and you can use internal resources to execute the POC.
   - You need to validate the POCs by seeking feedback from customers or potential customers and perform course-corrections if any.

2. Typically, it should not take more than 1 to 3 months to complete this phase. The KRAs that you need to accomplish in this phase are as follows.

KRAs

The KRAs for this phase are:

1. At the end of this phase, you would have a working POC that you can use to demonstrate the use of technology to solve a business problem to stakeholders including internal (CXOs) and external (customers) entities.

2. Another key deliverable of this phase should be a business strategy that elaborates as to how you'd like to use the CoE to grow the business multi-fold. This should include:

- What would be the key investments?
  i. People with specific skills: either hire them or train existing ones
  ii. Infrastructure: licenses for specific software or a specific hardware or anything else (e.g. various devices if you are going to build CoE for IoT)

- How long would it take before the organization would start seeing the benefit of investment? If you are setting up a CoE for any technology area, you should be patient enough to see these strategy bearing fruits. This period could be anywhere from 1 year to 1.5 years based on the technology area, market conditions and so on. However, it is important to have a common understanding amongst all stakeholders about the timeframe for which they can be patient!

- While you are working on a positive outcome, you should also be defining the cut off period for calling the CoE initiative off. Remember, your plan B is God's plan A. So, you need to also be ready for handling failures or setbacks.

3. Since you have spent some time in understanding the nuances of the technology well, this is also a good time to get some certifications under your belt so that it adds some credibility to your expertise. Most technology or platforms have some or the other partner program as well. For instance, your organization can become a Gold/Silver certified Microsoft partner by having some certifications and some customer references.

Walk the talk - AICC

You will have to build a POC to complete this phase and you will also have to come up with a business strategy here. It will be practically impossible to elaborate both of them in context of this article. However, I am providing some pointers below.

For building POC, you can check out the various scenarios mentioned earlier and use any of existing AI frameworks (DialogFlow or MS Bot Framework) and build the POC.

With respect to business strategy, you will have to spend time in thinking through this aspect well and build a strategy that includes the following:

1. Mission statement & goals
2. SWOT analysis
3. KPIs to track the success
4. Target customers
5. Competition analysis
6. Sales plan
7. Team composition
8. Operations plan
9. Financial projections

Phase 3: Jog

Pre-requisites

The key pre-requisites in this phase are as follows:

1. You should already have a team of people available who have some experience or knowledge of the technology. It should be a good mix of expertise and experience. So, you would have some key experts for a specific technology area and then you would train some people from your organization.

2. Since your CoE would be executing the projects in given technology, it is absolutely necessary that you have a plan in place that handles the scaling of operations well enough.

Jogging:

If you have reached this phase, it means that your CoE initiative has started to bear fruits in terms of business growth. Essentially, in this phase, you would have multiple projects running on the CoE offerings for different customers.

The key aspects to note are:

1. With your organization seeing a good traction on a specific technology area, you need to marshal your troops well in terms of creating seed teams and building satellite teams around them.

2. It is possible that you may get projects in specific sub-area of the technology that you have chosen.

In that case, you need to align your energies and resources to focus on that specific area. Let's say that you have started a CoE on Azure and you focused first on PaaS. However, you have customer projects/ requirements looking for IaaS offerings. In such a case, you need re-align your teams accordingly to handle that kind of requirement first.

In some variations of CoEs, it is in this phase when projects are handed over to execution units aka business units so that the CoE can focus on enhancing the competencies built and works pan-organization at times.

KRAs

The success factors for this phase are:

1. You now have a good enough knowledge about the technology since you have executed multiple projects. So, in a way, your CoE is like a well-oiled engine operating at full throttle.

2. While you are executing multiple projects, you are also required to keep on updating the offerings and case studies to ensure that the variety of work you are doing in specific technology area is captured and that can be shared with future prospects.

Jogging in AICC

In this phase, you have multiple AI projects already spanning across different AI platforms like DIalogFlow, Microsoft Bot Framework and so on. Since the core principles of AI platforms remain the same, you should be able to execute a project in some other AI platform as well.

Phase 4: Run

Pre-requisites

You are a champion now!

Key pre-requisites include:

1. You have significant knowledge and experience in a particular technology area now and you are able to build on that in order to be called as an expert in that area.

2. Your teams have worked on different projects and that has resulted into your CoE being rich in terms of versatility of specific technology areas. However, please do remember that technologies evolve. So, you have to continue sharpening your skills so that you do not fall behind when compared to competition.

Run

After having spent so much effort and time in building a CoE, in this phase, it is expected that you have the required skill and knowledge to be called upon as an **expert**. So, the CoE that you have built would have to make the following contributions:

Come up with best practices for specific technologies and drive them across organization

Have whitepapers/blogs written to demonstrate your technical prowess. In some cases, you may even apply for patents etc.

Work as internal consultant for the teams that are implementing projects in the same area so that they are more profitable to the organization.

KRAs

The key result areas that you would be fulfilling in this phase are:

1. You are being identified as a subject matter expert (SME) in a specific technology area.
2. You have a good understanding on the technology and authoritatively talk about best practices for executing projects in a specific technology area.

Running in AICC

For your AICC, you are being looked upon as AI expert and are appearing in various events. If that is happening, then you have achieved the goals.

Some Dos and Don'ts:

Following are some dos and don'ts when it comes to building and running CoEs based on my experience:

1. First and foremost, it is absolutely essential to have a complete buy-in from various stakeholders, right from the Jog phase. Any confusion or misunderstanding over the goals to be achieved could lead to failure of CoE's effectiveness. Communication plays a vital role in making this happen and it

is important to have periodic (at least monthly) meetings with stakeholders where data/details are presented about the CoE progress. This is when you can also discuss about gaps in understanding and have any course-correction done.

2. You need to be patient in the initial phases of CoE when you are building the competencies.

3. Usually CoEs work well when you have a core team assigned to it from day 1 and it stays on like that till the Jog phase. Frequent changes in this team would result into disruptions in the end results as well.

4. A CoE is a front-loaded initiative i.e. you have to first incur costs and then you realize the returns when things start happening for good. So, it is required that all the stakeholders are well aware of this aspect.

5. While you need to be patient, it is required that you are practical as well. You simply cannot endlessly keep on investing. You need to define your cut-off period in terms of realizing the investments on the CoE.

6. While choosing a technology, it is obviously expected that you choose one that has a good market traction. Neither should you be choosing something old or dated nor should you be choosing anything that is way ahead of time.

Candidates for CoE

Based on the current technology landscape, following are some candidates for CoE. Needless to state, this list is not exhaustive .

AI/ML: This one has to be on the top of the list.
Containerization technologies: Docker/Kubernetes
Test automation: keyword driven automation
JavaScript frameworks

Conclusion

You would have realized by now that building a CoE is not a child's play. However, with a good focus, commitment and dedication by right stakeholders, it is not very difficult to make it a success either. Depending on how you execute the blueprint, your CoE can be Canter or Cakewalk to Excellence.

The choice is yours to make!

Rahul Sahasrabuddhe
Author

Rahul Sahasrabuddhe has been working on Microsoft Technologies since last 17 years and leads Microsoft Technology Practice at a leading software company in Pune. He has been instrumental in setting up competencies around Azure, SharePoint and various other Microsoft Technologies. Being an avid reader, he likes to keep himself abreast with cutting edge technology changes & advances.

Thanks to Subodh Sohoni *for reviewing this article.*

Adam Storr

FAÇADE DESIGN PATTERN;

STILL RELEVANT IN ASP.NET CORE?

Part of the development process is constructing code to produce value to clients when the solution is delivered. To keep code clean, certain structures, or design patterns, should be embraced throughout the code base to aid with structure, maintainability, improve reuse and allow for easier team collaboration.

In this article I will look to explain one of the simpler patterns, how to implement it, and why it's still relevant in modern development in ASP.NET Core.

What is a façade?

To get an understanding of what the façade pattern is, we first need to look at where the name comes from and how it can be used.

As with a number of design patterns, the façade pattern takes its name from the construction industry and building architecture. A façade is defined as "the principle front of a building, that faces on to a street or open space" so it's the public face on multiple other parts of the building.

But what does this have to do with programming? Design patterns to the rescue.

Design patterns and well architected applications which follow development patterns and principles make life easier in the long run.

I've worked with a number of people in the past who have not seen the benefits of design patterns and just written code; it's usually worked but the code has been a bit of a mess!

I've also worked with several good developers who don't know what the patterns are called but if you describe them, they can relate and have seen/used them many times before.

Either way, design patterns will help you keep your code base readable and maintainable. The issue comes when to apply a pattern or just write the functionality directly and look to refactor in the future.

There will be times when you don't need a pattern or when you realise applying a design pattern earlier would have saved you time and headache. This unfortunately only comes with experience and working closely with more experienced developers.

There are also times when applying a design pattern because of what you "think" is going to change in the future and what you "expect" to come in the future is also detrimental to the quality of code as these events may not happen in your expected time scales or ever. This is where experience and refactoring come into the equation.

As with anything, when to apply a pattern or not comes with experience. When the need arises and if its beneficial to your codebase is a well-argued grey area. You have to decide for yourself.

Gang of Four

You can't talk about design patterns without a reference to "the" design patterns book. Often referred to as the "Gang of Four" book the full title is "Design Patterns - Elements of Reusable Object-Oriented Software". It is a highly recommended reference book. I'm still reading it, but I would say it's more reference material than a front to back page turner.

This book describes the façade pattern's intent as:

> "Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use"

What does a façade give me?

Like other design patterns, the façade pattern reduces tight coupling in your code to other parts of the system and the overall architecture as a whole.

Having a loosely coupled abstraction from the complexities of the subsystems allows for less complex testing and more maintainable code. The downside is that there are times when there will be more code files and lines of code which will feel like they should not be required.

It allows for a simplified interface on top of complex systems.

In the early development of a system this might not be an advantage as there are a number of moving parts especially with a multiple developer team working on the same code base. However, once the initial churn has settled, which with new code bases it will eventually happen, then additional functionality which needs to interact with the subsystems is usually easier and quicker as the consumer of the façade does not need to worry about the things "under the hood".

Not only do design patterns help with code quality but also with on-boarding new members of the team. Getting new members of a development team to be productive quickly is always a challenge, whether in a company setting or on an open source project. Keeping the complexities of the system initially abstracted away will allow for productivity to increase and avoid initial information overload.

So how do we implement it?

Façade Pattern - Simple Example

I hired a car on holiday recently and it had an interesting starting mechanism. From depressing the clutch and pushing the start button, it would execute a number of actions which would eventually lead to the car starting.

This is an example of a façade.

As a driver of the car, I would have to perform a couple of actions, but the car would have to do a lot more and interact with a number of subsystems to actually start the car. I have set out the functionality in a simple example below.

```
public class StartCarFacade : IStartCarFacade
{
  /** constructor and dependencies snipped for brevity **/
  public void Start()
  {
    // if the clutch is depressed then the sequence can start, if not then there is
    no point in continuing
    if (_clutchSubSystem.ClutchDepressed())
    {
      if (_engineCheckSubSystem.EngineReady()
      && _fuelSubSystem.HasFuel())
      {
        // start the engine
        _engineStartSystem.Start();
      }
```

```
            // and finally fold out the wing mirrors
            _wingMirrorSystem.FoldOutWinMirrors();
        }
    }
}
```

In the simplified example you can see the driver wants to start the car by pressing the Start button. If the clutch subsystem does not indicate that the clutch peddle is depressed, it will not start the car. If it indicates the peddle is in the required state, then it will continue and check the other subsystems; such as is the engine ready to start and is there enough fuel in the car.

My rental car also folded out the wingmirrors once the car had started. This is an example of a subsystem which is not necessarily related but still required to be actioned. The driver does not want to manually have to initiate this functionality every time themselves.

This is a good example of the Single Responsibility Principle (SRP) in action. Its sole responsibility is to start the car and orchestrate out the responsibilities of the subsystems to the subsystems. If the logic gets too complex, then it could violate the principle and some further refactoring should be done.

Higher Abstraction Level Example

You can use the façade pattern at various levels in your architecture depending on how it is designed and what type of application or system you are building.

Another good example of using the façade pattern is when you are working with a fundamental object in your system and it is used a lot. Not only is it used for the underlying functionality but when it is saved or loaded, there are a number of actions which need to be done to make sure it is in a valid state to allow for this.

This could be combining a number of different database calls to construct the required instance, different types of data storage option calls or even be reliant on external systems such as Azure Search, Azure Functions, SignalR or a message queuing technology. This allows for multiple communication techniques to be used and intra-microservice event notifications to be fired if required.

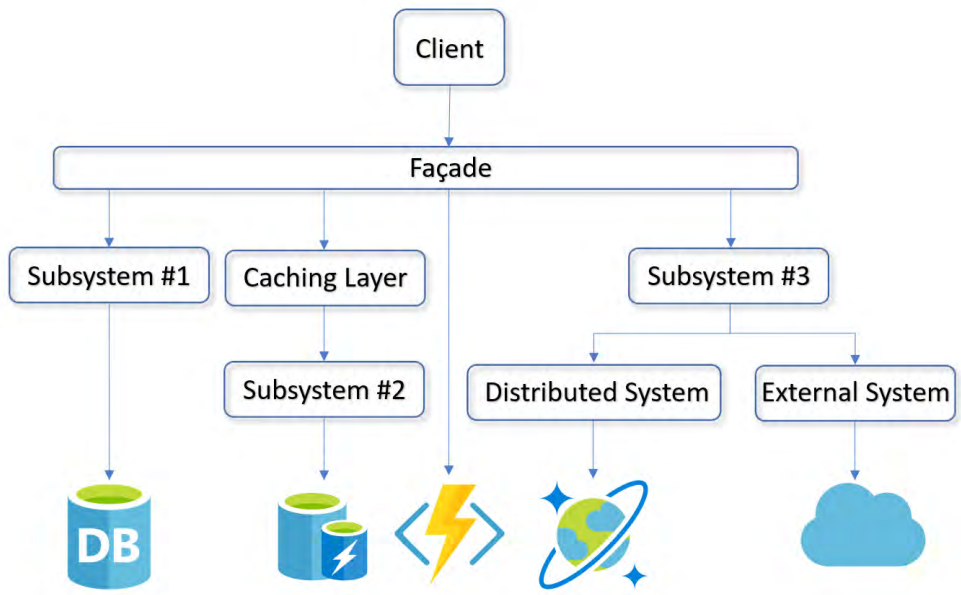


FIgure 1:

However at a higher level, it can seem that a service or façade has more than one responsibility and violates the Single Responsibility Principle. This can be the case, however it comes down to the correct level of abstraction you are using in your application.

In this example the façade is abstracting the other service calls and as before works as an orchestrator for the functionality. These can be complex requirements and integration points, but the façade is there to hide these away from the regular callers of the code which only want to read and write an object.

```
public interface IExampleObjectFacade
{
  Task<MyExampleObject> LoadAsync(Guid id,
  CancellationToken cancellationToken = default);

  Task<Guid> SaveAsync(MyExampleObject instance,
  CancellationToken cancellationToken = default);
}

public class ExampleObjectFacade : IExampleObjectFacade
{
  private readonly IMediator _mediator;
  public ExampleObjectFacade(IMediator mediator)
  {
     _mediator = mediator;
  }
  public async Task<MyExampleObject> LoadAsync(Guid id,
  CancellationToken cancellationToken = default)
  {
     return await _mediator.Send(
     new LoadMyExampleObjectEvent(id), cancellationToken);
  }
  public async Task<Guid> SaveAsync(MyExampleObject instance,
  CancellationToken cancellationToken = default)
  {
    return await _mediator.Send(
    new SaveMyExampleObjectEvent(instance), cancellationToken);
  }
}
```

The interface and implementation for the façade above defines high level `Load` and `Save` methods.

From the consuming code this is ideal as this is the right level of abstraction you want to use to read and write the instance your code flow is currently working on. Due to the complex nature of the actions required, the façade has the potential to become overloaded with dependencies and fall into the trap of becoming a "god" object.

In this example, I have used `MediatR` to delegate the processing to specific handlers for Loading or Saving. Please note that this could be seen as premature optimization, so be careful. Working with the methods on the façade then becomes straightforward. The consumer needs to know less about what is going on under the covers of the façade.

Note: *MediatR is an open source project which aids with implementing the mediator pattern. More information can be found on the Github page - https://github.com/jbogard/mediatr and on Wikipedia - https://en.wikipedia.org/wiki/Mediator_pattern*

Once the façade and its implementation have been created they will require registering with the ASP.Net Core DI mechanism. Like other services and library registrations this is done via the `ConfigureServices`

method in the applications Startup class.

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMediatR();
  services.AddTransient<IExampleObjectFacade, ExampleObjectFacade>();

  services.AddMvc()
  .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

Like any dependency, now you can specify your controller, or service, to require it through constructor dependency injection. Due to the way the façade is setup, you only require one dependency at this point instead of multiple references.

```
public class MyExampleObjectController : ControllerBase
{
  private readonly IExampleObjectFacade _facade;

  public MyExampleObjectController(IExampleObjectFacade facade)
{
  _facade = facade;
}

[HttpGet]
[ProducesResponseType(200)]
[ProducesResponseType(404)]
public async Task<IActionResult> GetById(Guid id,
  CancellationToken cancellationToken = default)
  {
    var item = await _facade.LoadAsync(id, cancellationToken);
    if (item == null)
    {
      return NotFound();
    }
  return Ok(item);
  }
}
```

In the simple controller class definition above, you can see that there is only one dependency and one action. As this controller grows there may be additional actions added to allow for reading lists or saving an instance of `MyExampleObject`, however no further dependencies would be required as all would delegate to the façade to perform the required actions.

In a more complicated application the power of a façade can be more prevalent inside interconnected services. This is especially the case when multiple functionality points require access to the items which are abstracted behind the façade.

Related Design Patterns

Leading on from the examples, the façade pattern works really well with other design patterns. In the example, it is using the Mediator Pattern which can help reduce the dependency graphs for the façade implementation.

Other examples of using different patterns with the façade pattern are as follows:

- For validation of an instance, or parts of an instance, which is being saved it might make sense to use the Strategy Pattern. The façade can take a dependency on the Strategy but not need to worry about any of the processing requirements which will be run. This allows for the validation to be updated without changing the façade itself. More info - https://en.wikipedia.org/wiki/Strategy_pattern.

- Related items which need to be generated or specific types which are required and known at runtime can be created using the Abstract Factory Pattern. This allows for related items to be instantiated without the façade requiring further dependencies. More info - https://en.wikipedia.org/wiki/Abstract_factory_pattern.

What are the downsides of the façade pattern?

One of the downsides, as previously mentioned, is it can result in a "god object".

So, what is a "god object"?

Like other services which evolve, the issue comes over time as the code base grows in functionality and complexity. As developers add small functionality points into the façade, additional dependencies are added without thinking about the bigger picture.

Initially it might be ok, but as typically developers' mentality is "find the previous example and copy it", this can balloon without care. Before long the dependency graph is huge, the method bodies are large and unwieldy and it generally becomes a mess.

There is also the potential risk that abstractions from different levels will leak across boundaries.

When adding in new functionality it needs to be made sure that it is being applied to the correct side of the façade. At the point of design, the functionality needs to be assessed with the potential for adjusting the level of abstraction. I've seen it done on both sides of the façade incorrectly and it can cause "leaks" which lead to developer headaches. It also diminishes the control which a façade can give you in your code base.

Making general sweeping comments about this topic is impossible, so make sure you discuss with your team, have peer reviews and pair programming sessions and talk it through to make sure it is applicable for your application.

So, is the façade pattern still relevant to ASP.NET Core?

In short; yes!

There are so many moving parts to applications in the modern world. Whether it is due to technology changes, data center location requirements for data storage or improved latency responses for customers or just complex processing, it makes the pattern applicable.

With the ever-increasing number of third-party services which are available, not to mention the new "serverless" wave which is currently being surfed, the potential is limitless. There will be new requirements for additional integrations, event sourcing, new caching techniques etc. All these can change and evolve

over time however the consuming code of the façade will not care about them as long as the functionality it expects remains consistent and the abstraction level is the same.

With distributed processing systems and multiple microservices requiring parts of the information to continue to process I feel it is relevant even more than ever.

Conclusion

The façade design pattern is a great abstraction when used correctly.

It will likely be used in conjunction with other patterns and practices to aid in reducing complexity, improving readability and keeping your code maintainable.

When should the façade pattern be used? Like most things in software development "it depends".

If it feels right, then it most likely is. If, however, you feel like you're fighting a losing battle then it's probably not the right pattern, the right level of abstraction or you need to evaluate the current state of your code.

The ability to apply design patterns comes from identifying an issue and refactoring the code to make it better.

Adam Storr

Author

Adam is a Technical Lead and Senior Software Developer with over 12 years of experience in various domain verticals primarily working with the Microsoft .NET stack. He enjoys investigating technical items and design patterns to aid with building scalable SaaS applications. Adam enjoys sharing what he has learnt in his blog - adamstorr.co.uk; concentrating mainly on .NET and ASP.NET Core. Adam's posts have been featured on ASP.NET as well as the ASP.NET Community Stand up. Adam lives in the UK with his wife and two children. He enjoys running and playing disc golf. Follow Adam on Twitter at @WestDiscGolf.

Thanks to Damir Arh *for reviewing this article.*

Sandeep Chadda

VSTS (VISUAL STUDIO TEAM SERVICES) IS NOW AZURE DEVOPS !!

WHAT HAS CHANGED AND WHY?

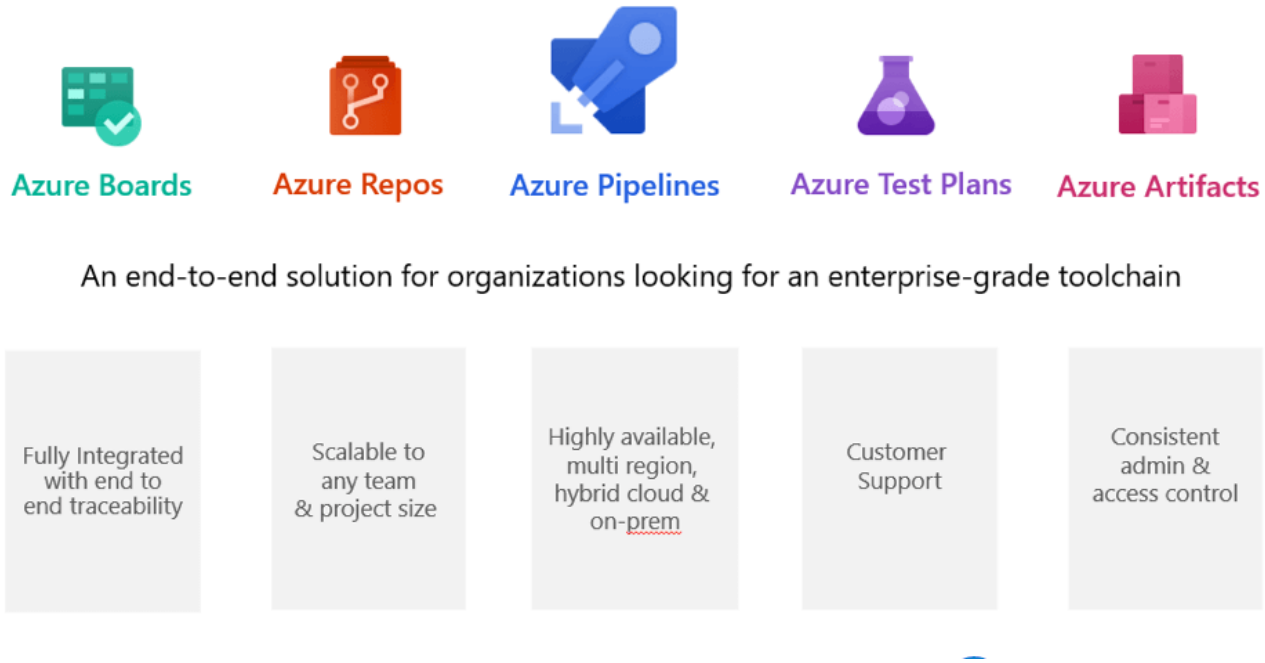
DevOps brings together people, processes and technology, automating software delivery to provide continuous value to your users. In other words, it enables any developer to ship customer value faster, more reliably, and with better quality.

Editorial Note: *If you are interested in a story of how 84,000+ Microsoft engineers moved to Visual Studio Team Services (VSTS) to create unparalleled value for their customers using DevOps, read Microsoft's Devops story.*

Azure DevOps

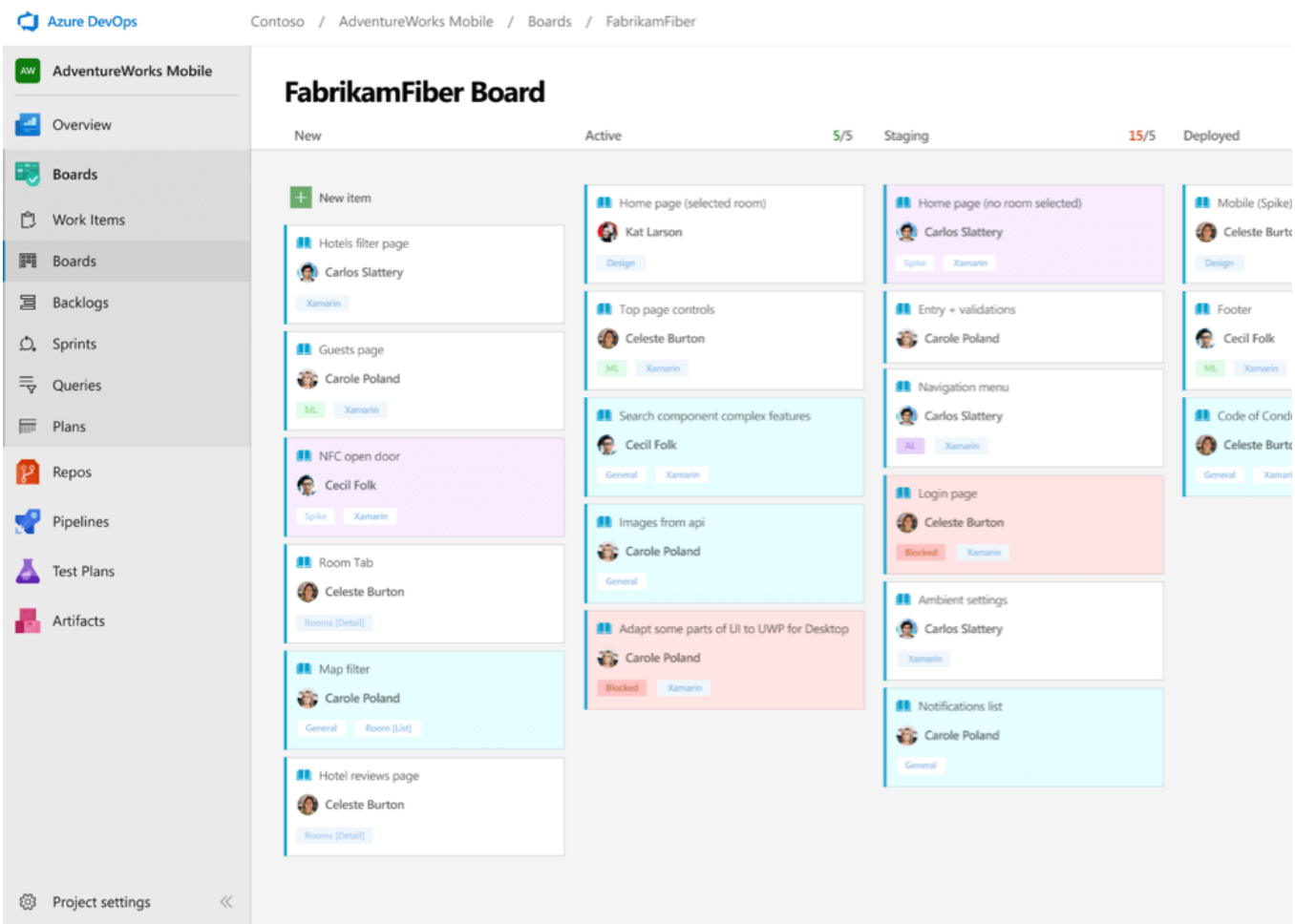
Azure DevOps (formerly known as VSTS) is everything you need to build your software product from beginning to end. Azure DevOps is a one stop shop that helps every developer on this planet to plan projects using Agile tools, manage code using Git, test the application, and deploy code using the best CI/CD system.

Azure DevOps comprises of 5 services that span the breadth of the development cycle. Let me delve deep into some of these services and then get to the *why* aspect of this change.



Azure Boards

Azure boards helps you to track work with Kanban boards, backlogs, and custom reporting.



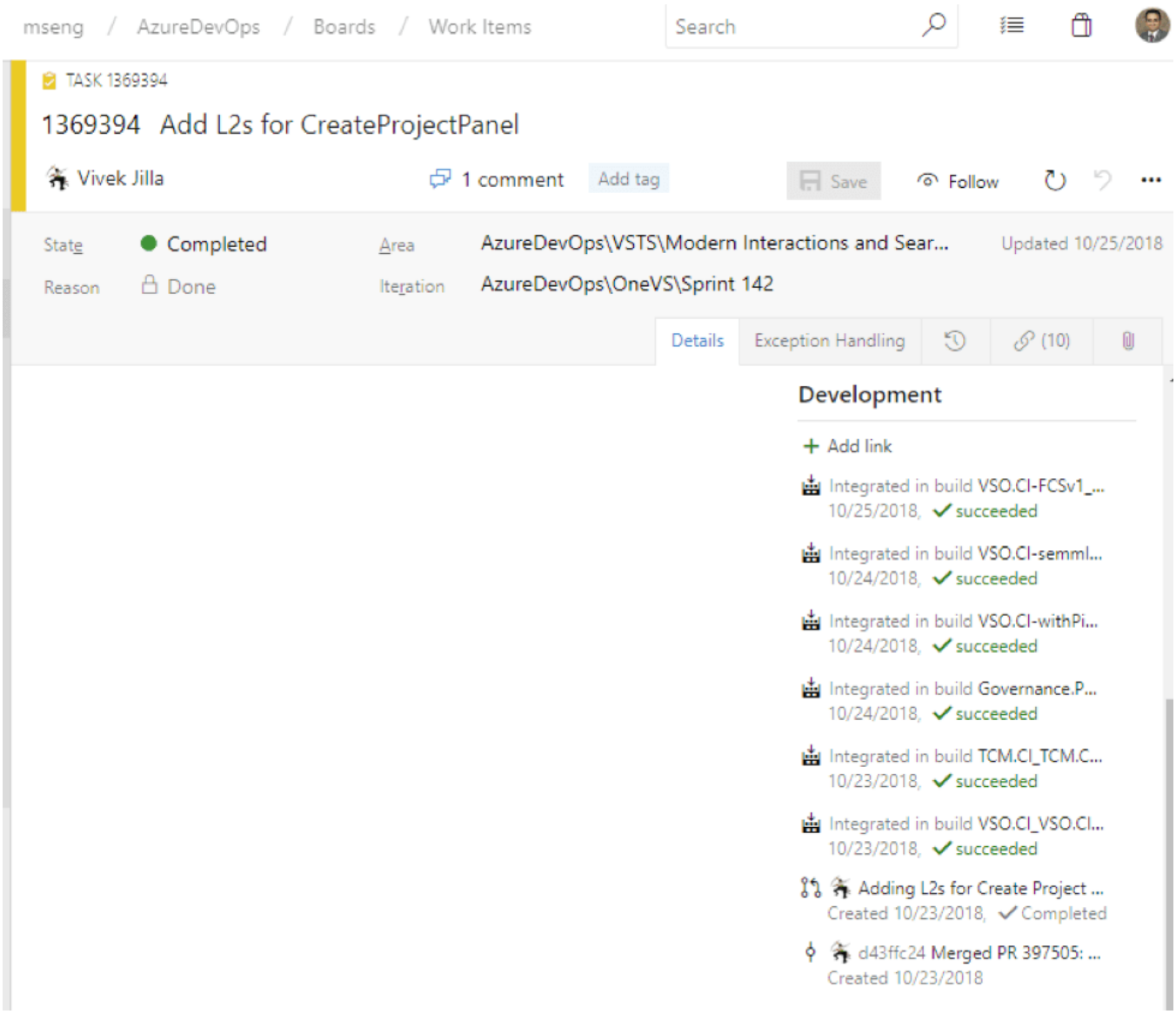
How we use it?

In Microsoft, the Azure DevOps team follows a 3-week sprint boundary where we ship at the end of each sprint. We have cross discipline teams of 10-12 people with clear charters and goals.

Teams use Kanban Boards to manage backlogs that ensure team autonomy while we use Plans to help us align with the leadership and the overall objectives of the product. We use queries extensively to track work and dashboards for real time reporting and tracking the work in progress.

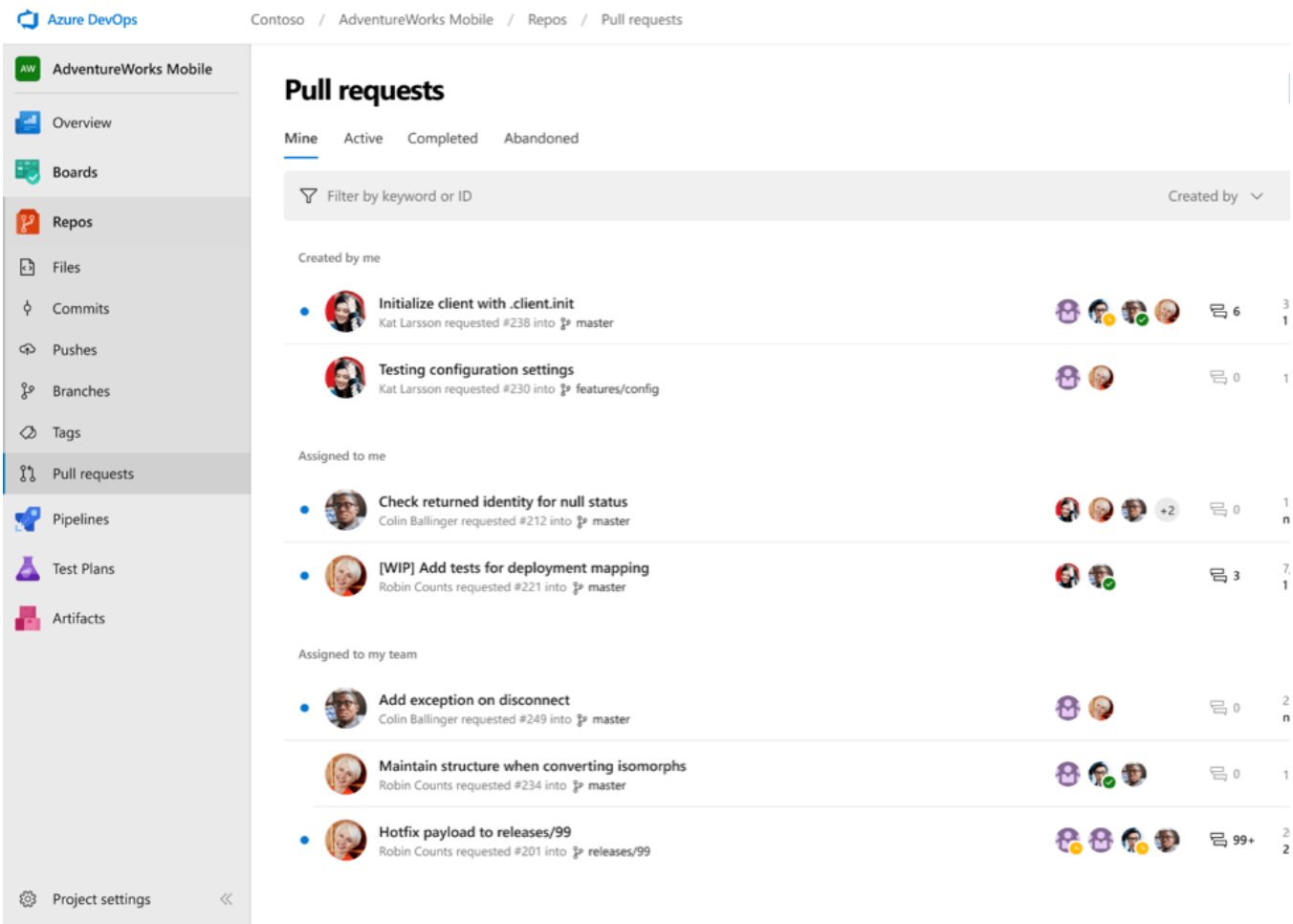
One feature that is a personal favorite is *traceability in work items*. i.e. Updates to the work item as it progresses through the dev and release cycle is automated thus ensuring that the status of a work item is always available and accessible to other stakeholders. Now I can view the commit, PR, build and release of a task from the work item itself.

Looking at this task below, I know that the commit was created on 10/23 – the PR was approved the same day and this fix was integrated in the build by 10/25.



Azure Repos

Azure Repos provides you with UNLIMITED Git repos. That means you can create a project in Azure DevOps and create as many repositories as you need and collaborate with your team members to build code with pull requests and advanced file management.



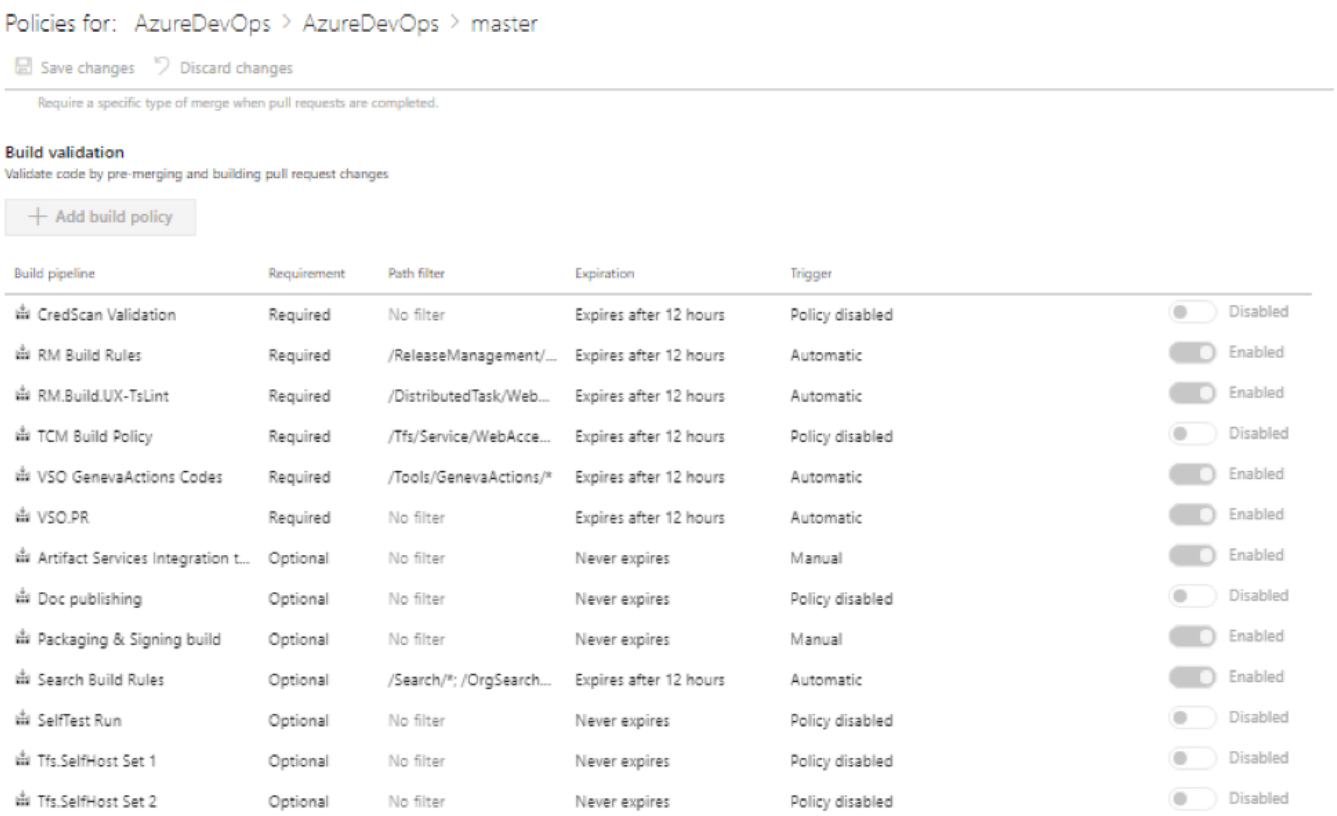
Microsoft enhanced git to scale to enterprise needs and invested in GVFS (Git Virtual File Systems) - https://gvfs.io/. GVFS helps in managing massive enterprise scale repositories. The Windows code base which is nearly 400GB in size is hosted on Azure Repos. A simple git clone command on the windows repo with git would take 12+ hours; but with GVFS it takes around 4-5 minutes.

How we use it?

In Azure DevOps, we use git in Azure Repos to maintain our code and use a combination of small commits, branch policies, PR reviews, and test with each check-in to ensure our code in master is always shippable.

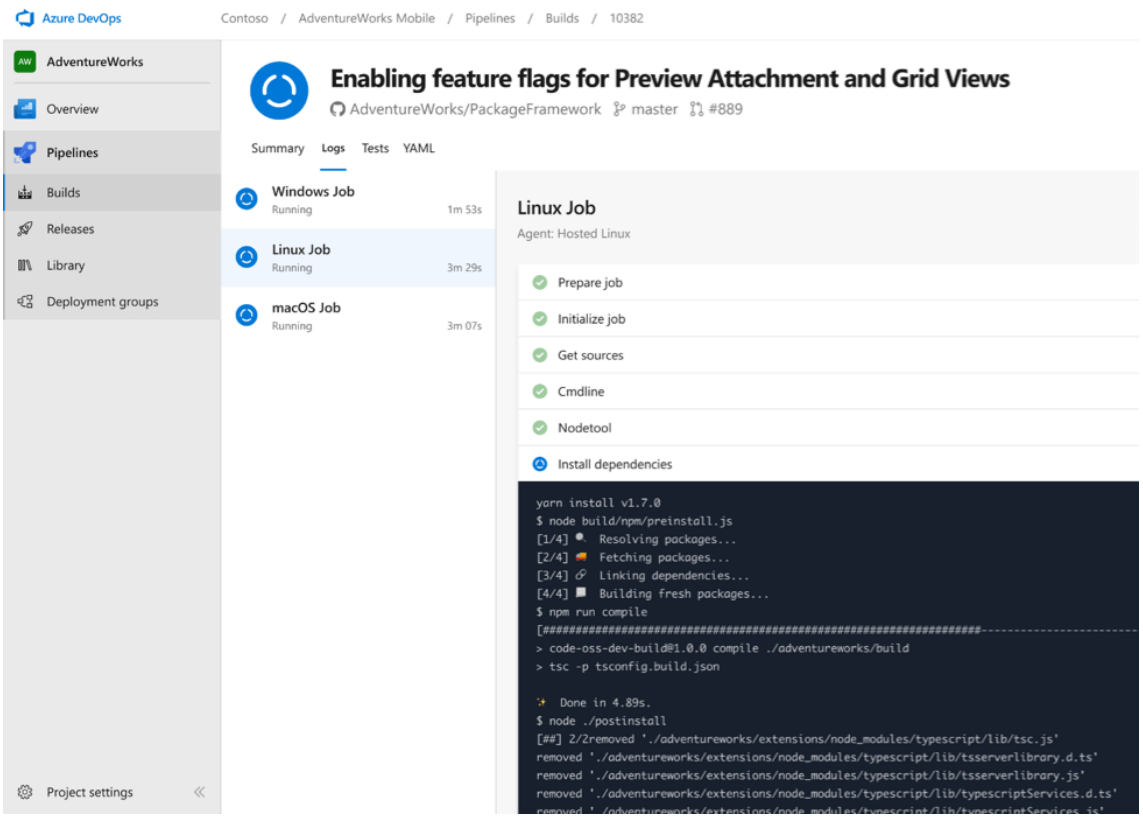
We work out of a single master, that helps us to eliminate merge debt. Considering that Azure DevOps has nearly 800 engineers, merge debt can be a potentially big problem. Using the pull requests acts as forcing factor to test and review our code, helping us to detect errors in the pipeline.

We run a bunch of tests with every merge to master thereby helping us ensure that master is pristine



Azure Pipelines

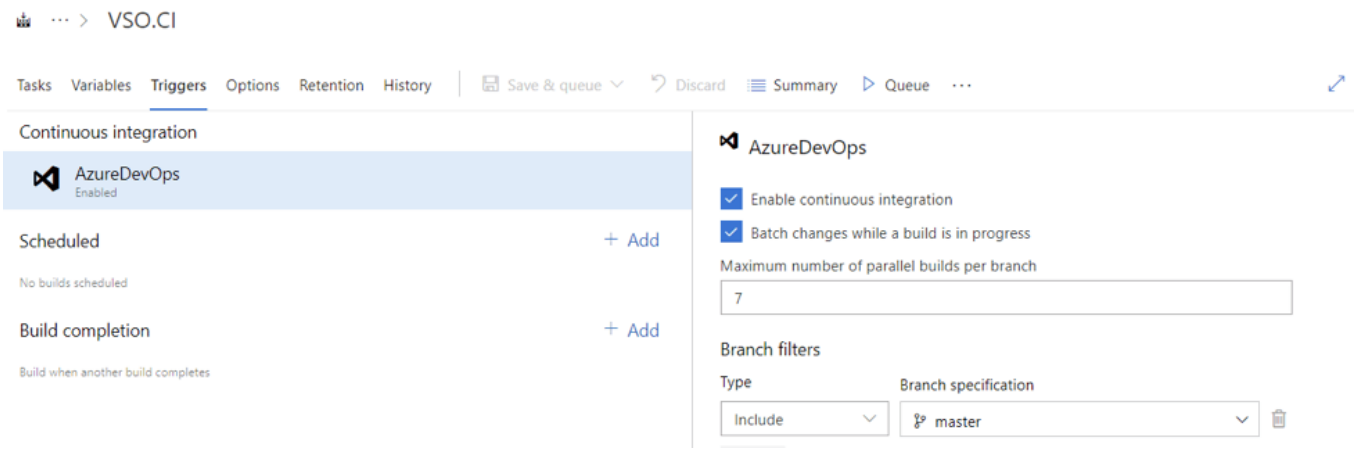
Azure pipelines allow you to build, test, and deploy with CI/CD that works with any language, platform, and cloud. You can use Azure Pipelines to connect to GitHub or any other Git provider and deploy continuously.



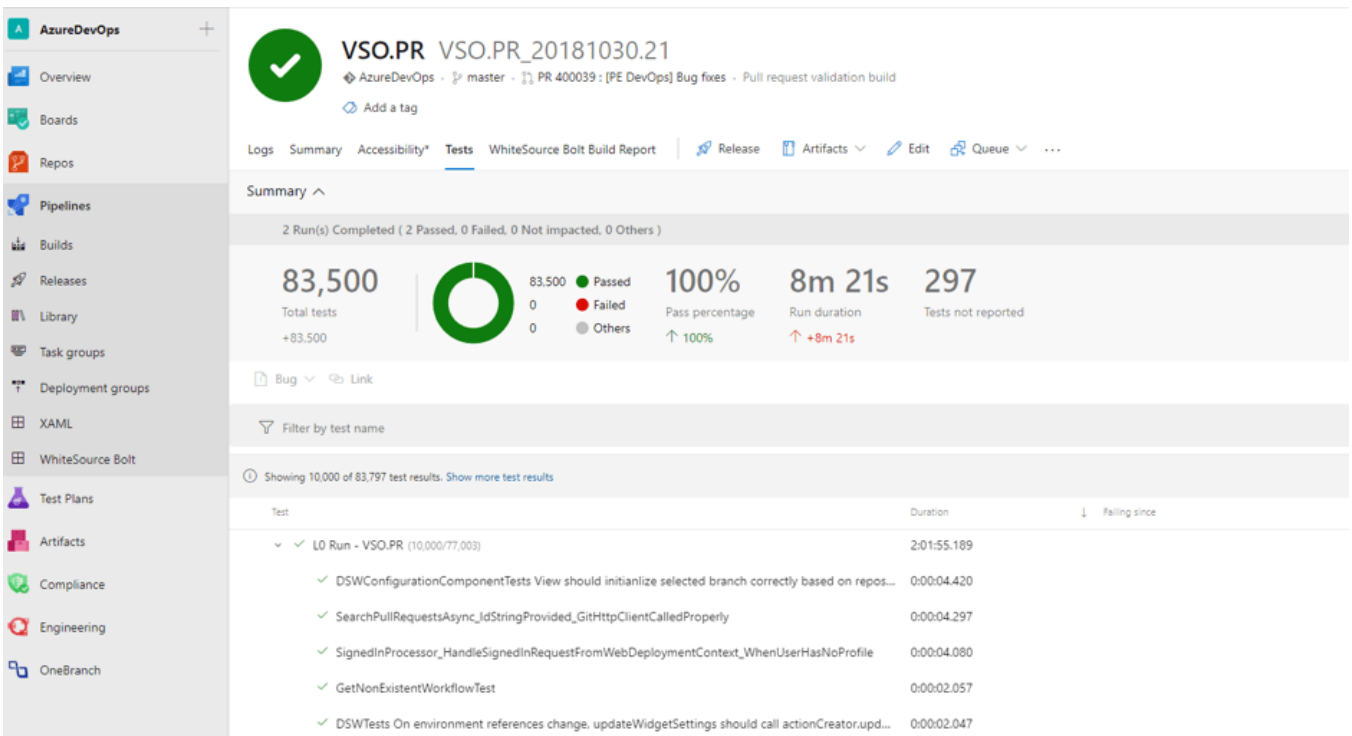
An interesting take on Azure Pipelines is that when it says any platform it means it. You can get cloud-hosted pipelines for Linux, macOS and Windows. It can help you to test and deploy Node.js, Python, Java, PHP, Ruby, C/C++, .NET, Android and iOS apps. To sweeten the deal, you also get unlimited minutes and 10 free parallel jobs for open source. With the latest support to push images to container registries like Docker Hub and Azure Container registry, it is a considerably powerful CI/CD solution in the market.

How we use it?

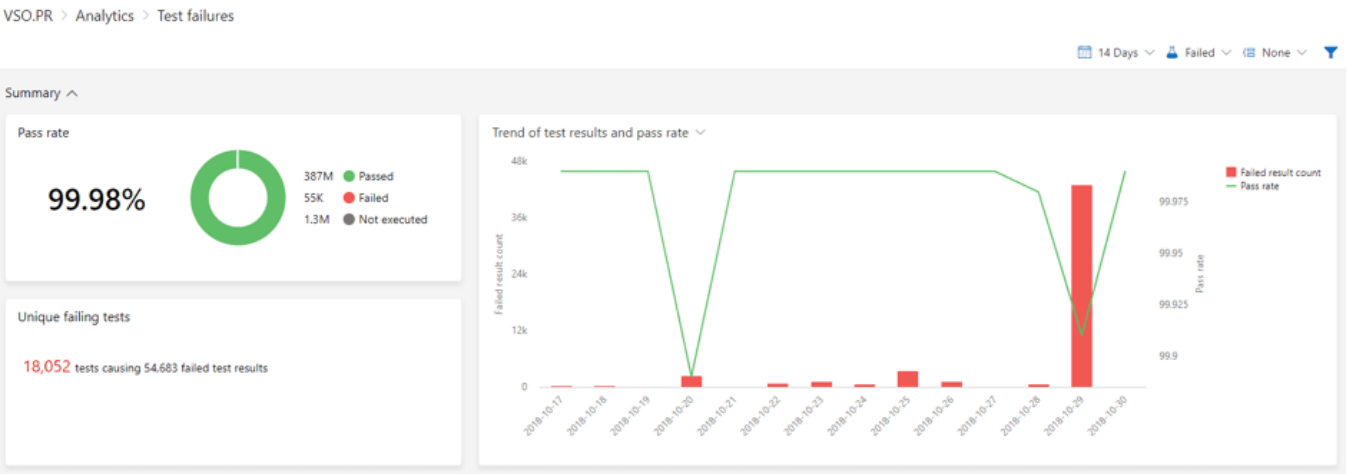
The Azure DevOps team ensures that a CI build is triggered for every check-in done into the master branch of the Azure DevOps git repository in master, which runs a bunch of unit tests.



Last I wrote on Azure DevOps in the month of march, we had 71k unit tests running with each PR. Now the new number is nearly 83,500 unit tests running in only 8 minutes. Measuring test failures is important for us and we ensure that there is a great debugging experience for test failure for developers to be productive.

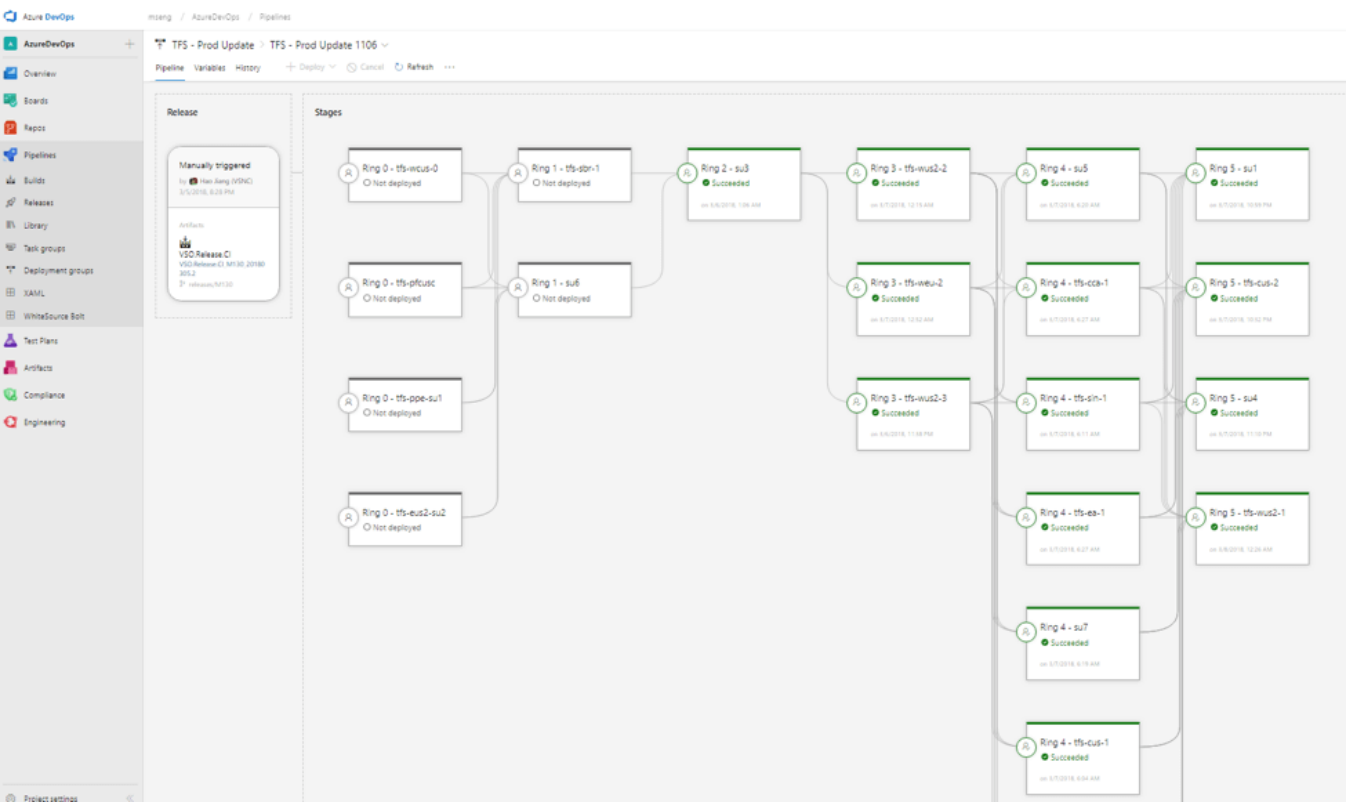


While debugging failures help address immediate issues, failure trends help us identify hidden patterns e.g. test that have failing continuously. We rely on Analytics to identify these patterns.



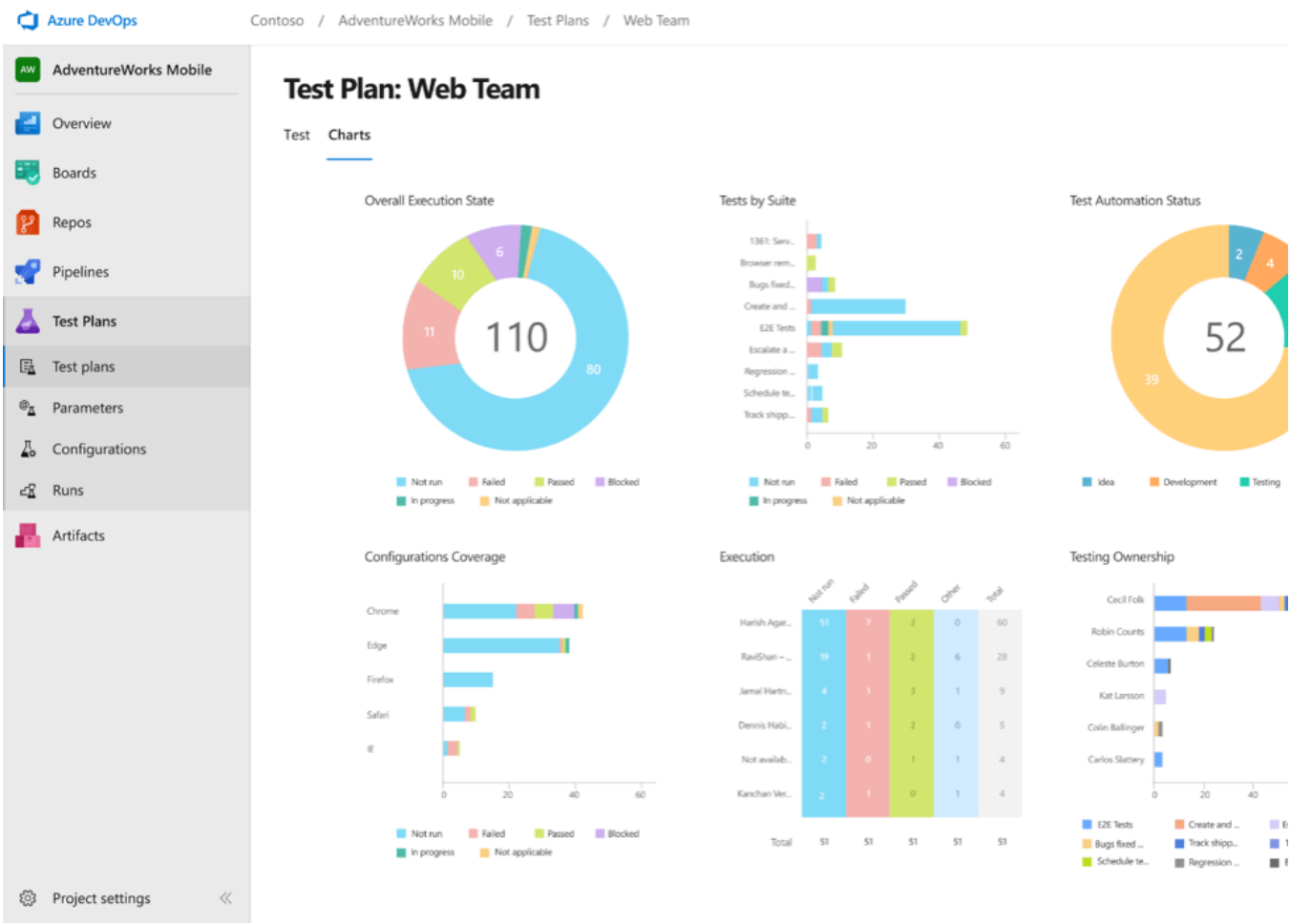
We also practice safe deployment i.e. stage-wise deployment of our products with adequate checks and balances in place to limit the blast radius of any impact due to an issue during deployment.

Safe deployment practice adopted by Azure DevOps team to deploy Azure DevOps to its users in stages



Azure Test Plans

You can use **Azure Test Plans** to run manual tests for your web and desktop applications and also log defects. . You can track and assess quality throughout your testing lifecycle that help you to get end to end traceability.



How we use it?

In Azure DevOps team, every engineer is responsible for code and test. We call this as combined engineering which improves accountability and incentive for developers to write better tests and more automation. We stringently adopted a three fold mantra to ensure quality in code:

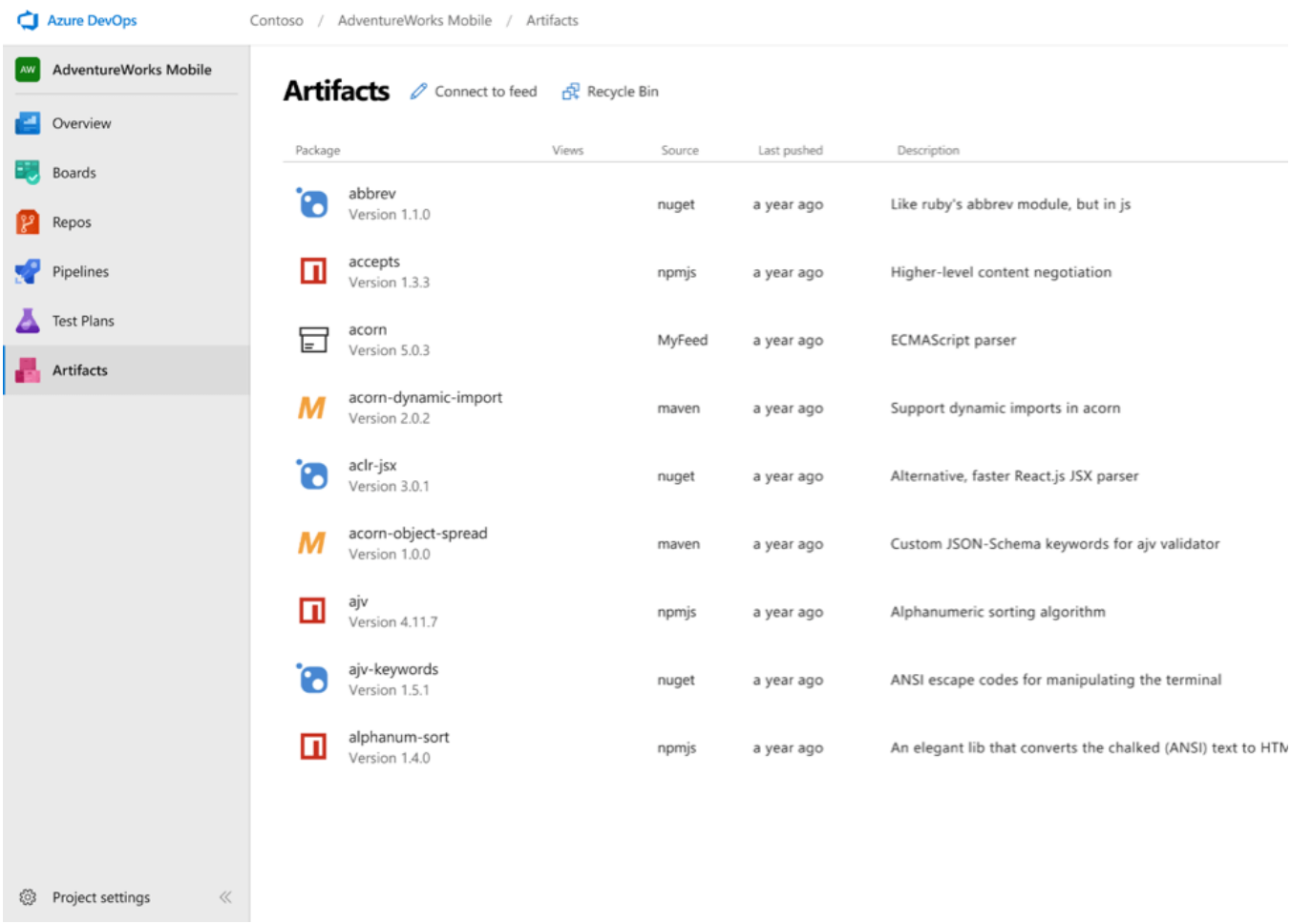
1. Developers owns code quality

2. Master is always healthy and shippable. Shift-left for testing i.e. greater emphasis for unit tests and eliminate flaky tests

3. Shift right in production i.e. Ensure quality in production by rolling out changes in a progressive and controlled manner. Also do fault injections and chaos engineering to see how the system behaves under failure condition.

Azure Artifacts

Azure Artifacts helps you to create and share Maven, npm, and NuGet package feeds from public and private sources – fully integrated into CI/CD pipelines.



How we use it?

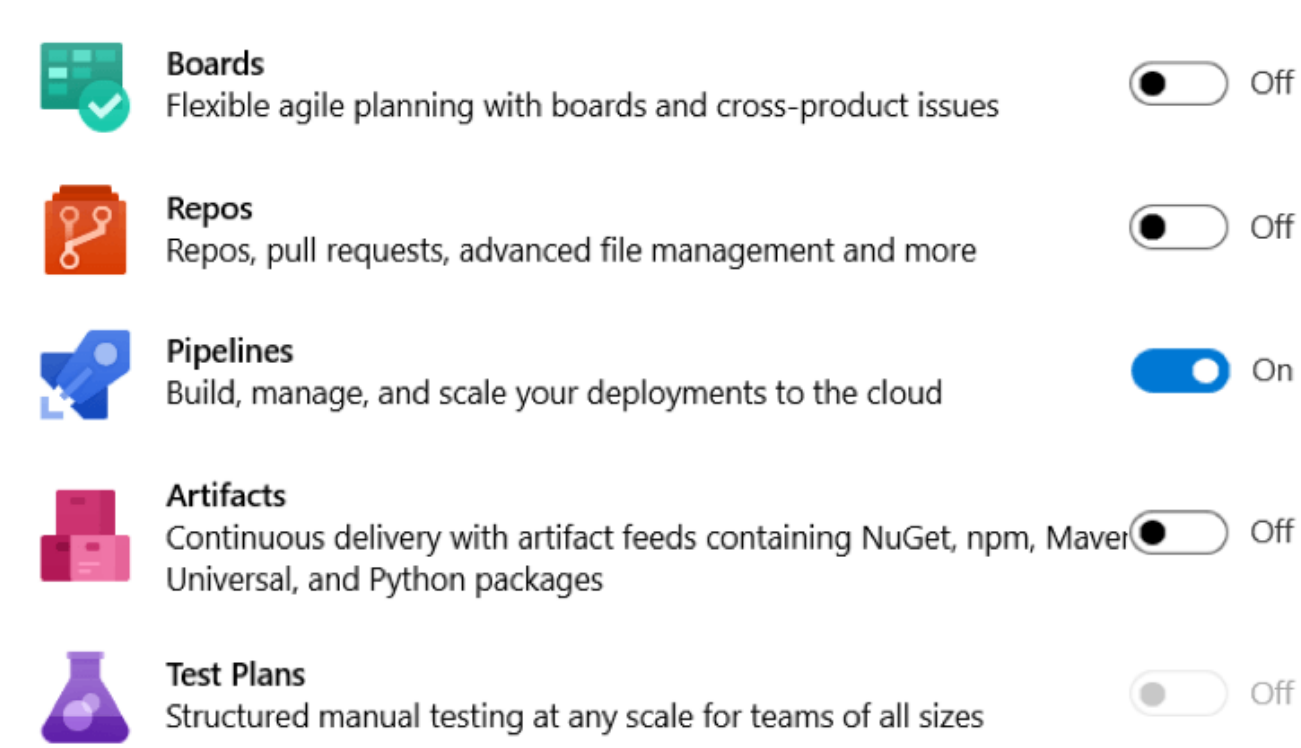
Azure Artifacts pretty much completes our offering end to end. It is now a one stop shop for managing universal packages. It is the backbone for inner sourcing in Microsoft where-in a team in Windows creates a package that is shared across teams and it becomes a source of consumption.

Why Azure DevOps?

We made this change since it was just the right thing to do for our customers. While Azure DevOps is an integrated suite to provide end to end DevOps capability for large enterprises, it is also nimble for organizations who want to use only few services e.g. you can host your code in a git repository in GitHub and use only Azure Pipelines for build and deploy.

Azure DevOps now allows you to select the services that you want to use for your organization.

Select the service that matter the most to you in Azure DevOps



Microsoft is steadfast to make Azure DevOps, the best end to end DevOps offering and would love to hear from you on how they can make it better.

Suggestions: Post your suggestions to help Microsoft improve Azure DevOps on their dev community.

https://developercommunity.visualstudio.com/spaces/21/index.html

Special thanks to all the reviewers mentioned below.

Sandeep Chadda

Author

Sandeep is a practicing program manager with Microsoft | Visual Studio Team Services. When he is not running or spending time with his family, he would like to believe that he is playing a small part in changing the world by creating products that make the world more productive. You can connect with him on twitter: @sandeepchads. ***You can view what he is working on right now by subscribing to his blog:*** aka.ms/devopswiki

Thanks to Divya Vaishnavi, Jaiprakash Sharma, Mahathi Mahabhashyam and Vinod Joshi for reviewing this article.

Keerti Kotaru

BUILDING SINGLE PAGE APPLICATIONS WITH ANGULAR ROUTER

Routing in Angular enables navigation from one page to another while the user performs application tasks. Routing also allows you to define the modular structure of your application. However, there is much more to routing than just moving the user between multiple views of an application.

In this Angular Routing - Basic to Advanced tutorial, we will learn:

Part 1

- *basics of routing with Angular*
- *the need for routing*
- *getting started instructions to add Routing to an Angular project*
- *configurations for routes*
- *creating links that perform view transitions*

Part 2

- *asynchronously resolving a route after obtaining data*
- *conditionally redirecting to an alternate route*
- *authorization checks on a route etc.*
- *and more.*

ANGULAR ROUTING – BASIC TO ADVANCED

Off late Single Page Applications (SPAs) have become a norm with rich web UI applications being built as SPAs.

SPAs have a unique advantage that the user doesn't lose context between page transitions. Yester-year applications reload the entire page as the user tries to navigate between views. The flicker not only affects user experience, it is also inefficient. An update to a section of the page needs the whole page to reload. SPAs address this problem.

Imagine a travel website. One of the screens show a list of destinations. The User clicks on a link in the list and the whole page loads to show the details although the Page header, footer, navigation controls (like breadcrumbs, left nav) etc. haven't reloaded at all. They might have been updated to show the latest content but they haven't reloaded the whole view. This leads to a better and rich user experience.

Editorial note: *A way to circumvent the entire page reload problem is to use AJAX which would allow the client to communicate with the server, without reloading the entire page. However, a key difference between SPAs and AJAX is that SPA is a 'paradigm' for how to develop web apps and it may or may not use AJAX to achieve its goal. AJAX on the other hand is a 'technique' about communicating with the network and server in the background.*

Angular provides router API to manage a SPA. The Angular Router takes care of the following.

- Allows a URL to represent a view in the application. For example, *my-travel-website.com/destinations* represents a list screen. And *my-travel-website.com/destination/hyderabad* represents a view that shows detailed information about the travel destination Hyderabad.

- Create links that can preform view transitions. In other words, navigate from one view to the other.

- Allow creating deep-links to specific pages in the application. Without routing and URL tied to a view, the user doesn't have a way to deep-link to a screen and would need to go through the screens before it, every single time. User can't bookmark or save a link the page. Routing solves this problem.

  For example, if user wants to get to the details screen for London, he/she may navigate directly to *my-travel-website.com/destination/london*. The user doesn't need to navigate to the list screen and subsequently click on London.

- It is convenient for users to go back and forward using browser buttons. Routing ensures, it doesn't take away the ease of using browser back and forward buttons or actions.

PART 1- GETTING STARTED WITH ANGULAR ROUTER

If you are using Angular CLI for getting started with a new Angular project, use the following command to include routing.

```
ng new my-travel-app --routing
```

The `--routing` flag indicates routing to be added to the solution. It does the following to add routing features.

1. Install Angular router from the MonoRepo. To install it manually use the following command
   `npm install -S @angular/router`
2. It's preferred to separate routing logic to an Angular module of its own. Hence, a module named `AppRoutingModule` is created in a file `app-routing.module.ts`
3. Import `Routes` and `RouterModule` from `@angular/router`
4. Configure routes. Next section explains route configuration in details.
5. Provide route configuration as a parameter to `forRoot` function on the RouterModule. Import the returned module.

```
@NgModule({        // routes has the route configuration.
// It is a variable of type Routes
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Also note that the RouterModule is exported. It allows the routing to be consumed in the root module where the routing is used.

Notice the code snippet in app.module.ts. It is the root module in the sample application.

```
import { AppRoutingModule } from './app-routing.module';

@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

ROUTE CONFIGURATION

As described earlier, SPAs use routing to tie a view to a URL (which is nothing but a route). Route configuration defines route pattern (or the URL pattern). It ties a component for a given route pattern. A Component renders the view.

A route configuration object starts with two basic fields:
url – route pattern
component – the component to render when a route pattern matches.

Any application will have more than one routes to configure. Hence route configuration is an array of objects. Consider the following routes.

```
const routes: Routes = [
  {
    path: “destinations”,
    component: DestinationListComponent
  },
  {
    path: “destination/:city”,
    component: DestinationDetailsComponent
  }
];
```

In the sample application detailed in this article, we have two views

1. List screen, showing list of destinations to travel. When the URL matches ***destinations*** (see Figure 1), `DestinationListComponent` is rendered. It shows a view for list of cities.

2. Details screen, showing a detailed view of the selected city.

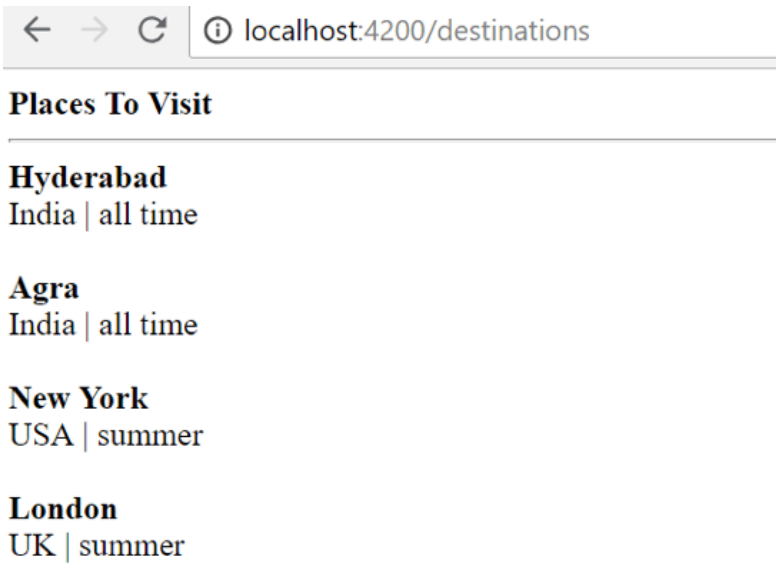


Figure 1: List screen

Notice that `city`, is a route parameter. Hence, it's qualified with a colon. The `DestinationDetailsComponent` can read the value and query details.

Mandatory route parameters

Traditionally web applications shared data between two pages in the URL.

Sections of the URL might define data required for the route. In the above example (/destination/:city), when the user navigates from list screen, the selected city is passed on as a route parameter. Imagine that the user selected Hyderabad. The URL `/destination/:city` translates to `/destination/Hyderabad`. The URL has the city name, which is nothing but the data. Here, the data, city name is very much part of the URL pattern.

Now we will have to use the data provided in the URL. The Component can access the URL parameter to query details about a particular city. Consider the following code snippet.

```
// ... For brevity show only the lines of code relevant
import { ActivatedRoute, ParamMap } from '@angular/router';

@Component({
  // ... For brevity show only the lines of code relevant
})
export class DestinationDetailsComponent implements OnInit {

  private destination: string;

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    this.destination = this.route.snapshot.params.city;
  }
}
```

The statement `this.route.snapshot.params.city` retrieves route parameter city's value. In the example destination/Hyderabad, the value is Hyderabad.

Notice `ActivatedRoute` is imported and injected into the component. It refers to the route associated while loading the component. It has a field snapshot, which has details about the route at that point in time. It is of type `ActivatedRouteSnapshot`.

One more level down, the child field params are key value pairs of data. They have all the route parameters for the given URL. In the above example, city is the only route parameter. Hence, we can retrieve the value of the parameter with variable name provided in the route configuration (city).

You may use the city value to query details of the city and show it in the details screen. See Figure 2.

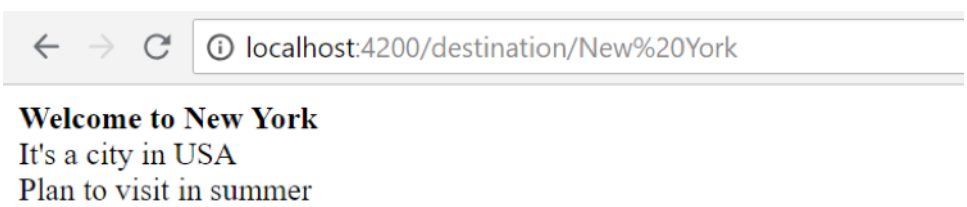


Figure 2: Details Screen

Optional route parameters

In a different scenario, data could be optional. Just to tweak the example a little bit, imagine `/destination` would show the first city in the list. To show details of a specific city, provide city name in the URL. In this case, city will become an optional parameter. The prior approach with city name as a route parameter wouldn't work because city name is part of the URL pattern.

If the city name is not provided, the pattern doesn't match. Hence, we would add a new route configuration to support optional city name. For clarity, let's name the new route `destination2`.

Note: It need not be given a different name destination2. We may continue to name it as `destination` and a different configuration object without the mandatory city name in the route.

```
{
  path: "destination2",
  component: DestinationDetailsComponent
}
```

As a convention, in a URL, optional values are supplied with Query parameters (query strings). The URL could be `/destination2?city=Hyderabad`. Or it may just be `/destination2`, in which case we will show the first city details. City is not part of route configuration anymore.

The component can now retrieve city name from `queryParameters` field instead of `params`. Refer to the following line of code for city name from query parameters.

```
this.route.snapshot.queryParams.city;
```

Notice we are using the same component `DestinationDetailsComponent` for both the routes (destination and destination2). In the example, the component needs to handle retrieving details from the route param or a query param.

Consider the following code snippet:

```
// ... For brevity show only the lines of code relevant
import { ActivatedRoute } from '@angular/router';

@Component({
// ... For brevity show only the lines of code relevant
})
export class DestinationDetailsComponent implements OnInit {

  private destination: string;

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    // Get city name from params or query params
    let cityName = this.route.snapshot.params.city || this.route.snapshot.
    queryParams.city;
    if(cityName){
      this.destination = cityName;
    }else{
      // Get first city name from the list.
    }
  }
}
```

Optional fragments in the URL

It is a common practice in URLs to use breakpoints or hash fragments. They qualify a section of the page or view.

Let's further alter the destination details view in the code sample. When no optional parameter was specified instead of always showing the first value, we may use an optional segment to decide first or last.

Access the fragment values using the following code statement. On the snapshot object, retrieve value from the `fragment` (like `param` or `queryParam`):

```
let fragment = this.route.snapshot.fragment;
```

Subsequently, pick the last city from the list if the segment value is last. See the code snippet and Figure 3.

```
this.destination = cityObjects && _.isArray(cityObjects) && ((fragment && fragment
=== "last") ? cityObjects[cityObjects.length-1] : cityObjects[0] );
```

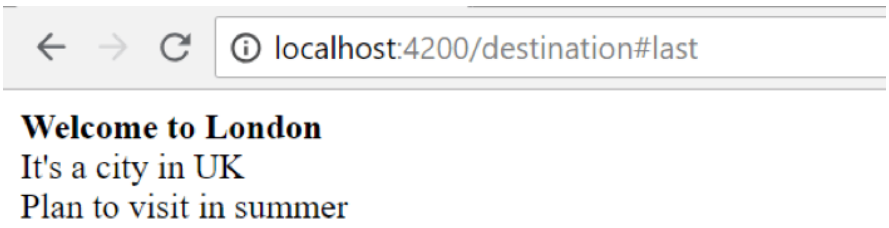


Figure 3: Show last city in the list when fragment value is last

The sample component will show the first city details (in the list), when no fragment or query string is provided.

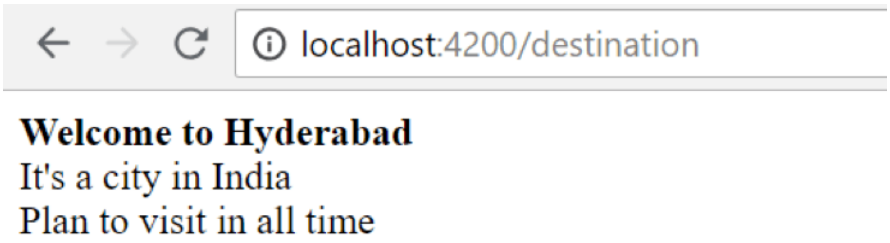


Figure 4: URL without any optional params

Additional Route Configurations

So far, we have seen three route configurations, one for destination list screen and two more for destination details screen.

It's common to have a default route, which will be used when there is no route specified (neither destinations nor destination/[city name]). Imagine we have decided to make the list screen as the default route when no value specified.

Consider the following configuration object

```
{
  path: “”,
  redirectTo: “/destinations”,
  pathMatch: “full”
}
```

When path is an empty string, we redirect to `/destinations` (the list screen). The field `pathMatch` qualifies the configuration and redirects to the destinations route when the given path matches completely.

In this case, it's an empty path.

Whenever there is no path after domain name, it defaults to destinations route.

We may use `pathMatch` value `prefix`. It would qualify the configuration when the given path includes path specified in the configuration. Consider the following snippet.

```
{
  path: “list”,
  redirectTo: “/destinations”,
  pathMatch: “prefix”
}
```

It matches any path that includes list at the beginning of the route. Above configuration matches the path `/list/cities`. However, it doesn't match if the list is not at the beginning. For example, `0 or /test/list/cities` doesn't match.

Please note for an empty string (as the path), which is default route for the application, `pathMatch` value should be `full`. If `prefix` is used, it will match every URL. It's an empty string after all.

Router Outlet

Router-Outlet acts as a placeholder for the component at a given route.

Routing is applied on router-outlet.

In a typical application, a root component will contain the router-outlet. There could be other components on root component beyond the router outlet. They could be header, footer, navigational controls like side nav etc.

See Figure 5 which depicts an application skeleton. Notice the Destination List component next to left nav, between header and footer. Routing is applied here as the router-outlet is positioned here. As route changes, a component configured for the given route takes place of the router-outlet.

In this case, as the route is /destinations, list component is shown on the outlet.

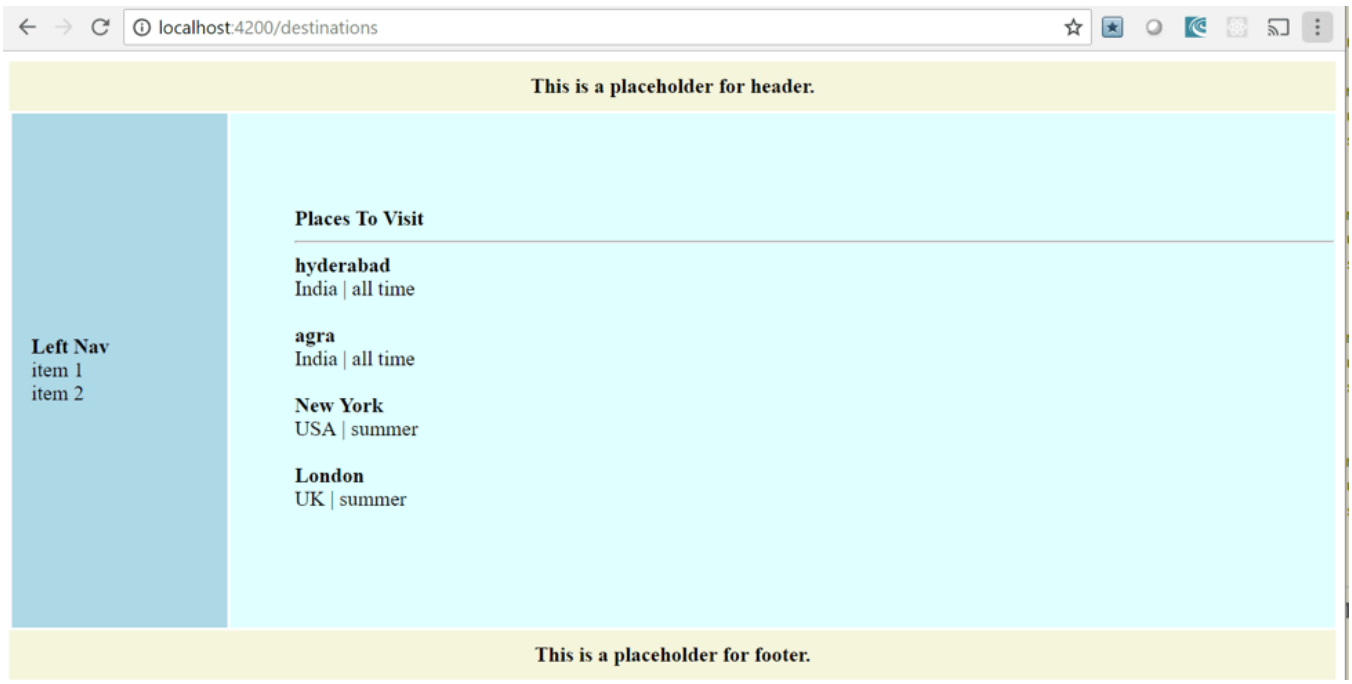


Figure 5: Application Skeleton

See Figure-6 with details screen in place of router-outlet:

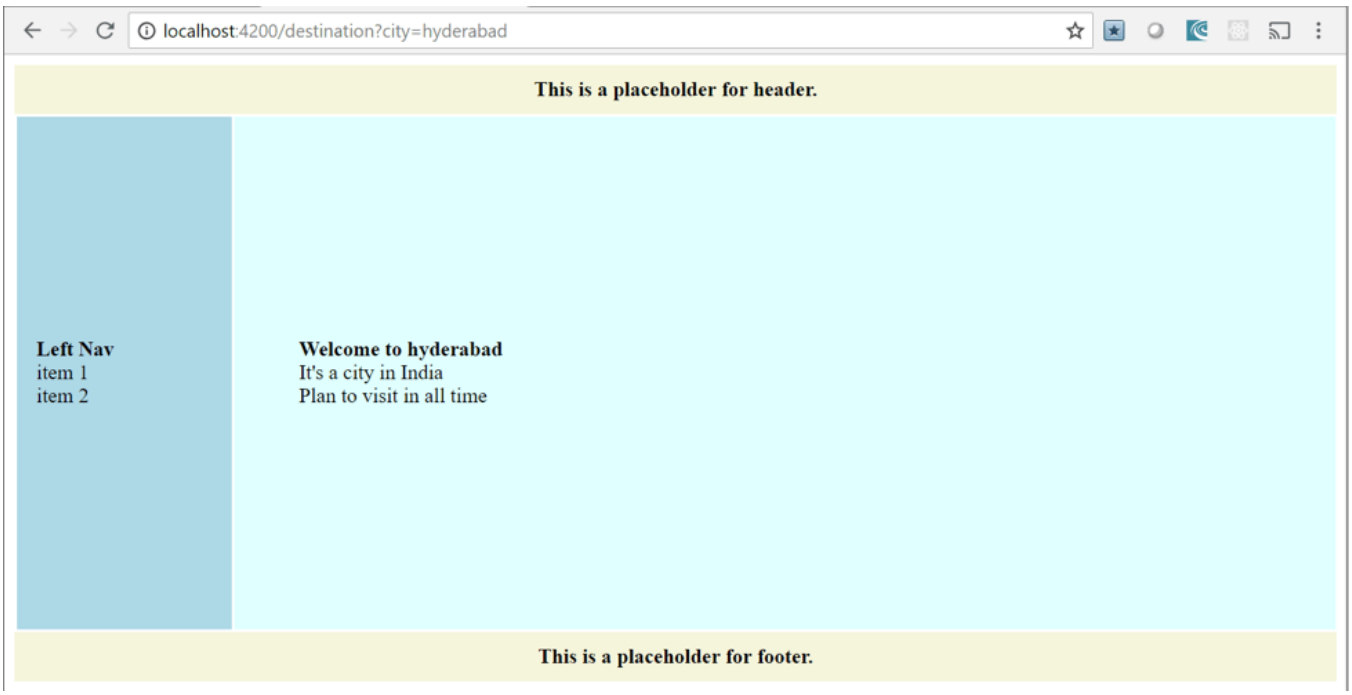


Figure 6: Details screen on the router-outlet

Consider the following code snippet. It's the root component (app.component) template (html file). Notice the highlighted router-outlet. As route changes, a component is rendered at this place.

```
<div style="background-color: beige; padding: 10px 10px 10px 10px; text-align:
center">
 <strong> This is a placeholder for header. </strong>
</div>

<table style="min-height: 400px">
 <tr>
  <td style="background-color: lightblue;min-width: 150px; padding-left: 15px">
```

```
   <div >
    <strong>Left Nav</strong>
    <div>item 1</div>
    <div>item 2</div>
   </div>
  </td>

  <td style="padding-left: 50px; background-color: lightcyan; min-width: 800px;">
   <router-outlet></router-outlet>
  </td>
 </tr>
</table>

<div style="background-color: beige; padding: 10px 10px 10px 10px;  text-align:
center">
 <strong> This is a placeholder for footer. </strong>
</div>
```

Router Link

We have almost come to the end of Part 1 of this article just by describing the creation and configuration of routes which renders a view or a component based on a URL (route). **However, how do we link to a route in a SPA built with Angular?**

We will use Router Link. It's a directive part of `@angular/router` module.

Use router on an anchor or a button element. Consider the following code snippet:

```
<div> <a routerLink="/destinations">Home</a> </div>
```

Here the *home* link takes the user back to list screen. However, the route is static, so to create a dynamic link, consider using the directive as follows.

```
<a [routerLink]="['/destination', dest.name]"> <strong>{{dest.name}}</strong></a>
```

In this case, the directive is concatenating array of values to create a dynamic route. The dynamic value includes name of the city.

Following are more input attributes used often with Router Link.

Query Params:

To the `queryParams` input attribute, provide a JSON object representing key value pairs on the query string.

In the following example, the value will be the selected city. The anchor element with the routerLink directive creates a link, clicking on which the user navigates to a route shown in Figure 7.

```
<a [routerLink]="['/destination']" [queryParams]="{cityName: dest.name}" >
<strong>{{dest.name}}</strong></a>
```

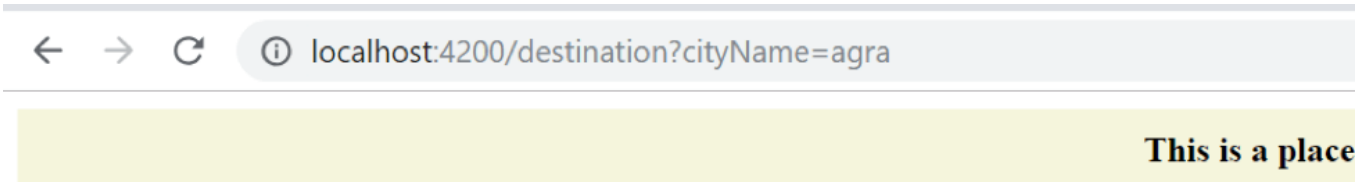


Figure 7: Query string created based on input attribute queryParams

We can pass multiple key value pairs in the object for multiple query strings. The following code snippet creates a link, clicking on which, the target route will have multiple query string separated by ampersand (&). See Figure 8 for the target route.

```
<a [routerLink]=”[‘/destination’]” [queryParams]=”{debug: true, cityName: dest.
name}” > <strong>{{dest.name}}</strong></a>
```

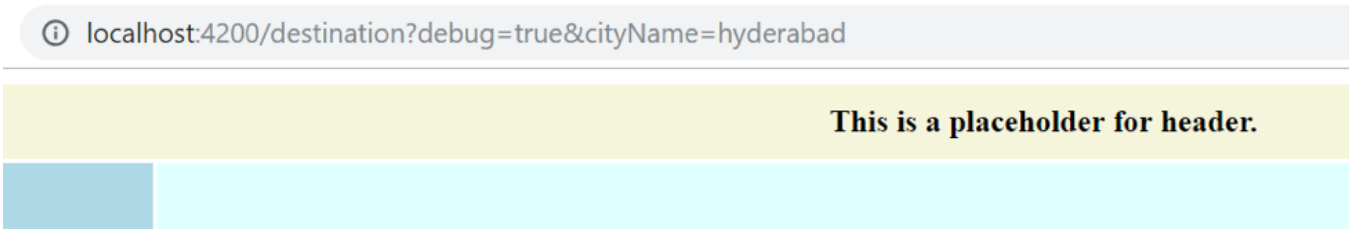


Figure 8: Multiple query strings

Fragments

Pass fragments in the link with fragment input attribute. Consider the following code snippet. See Figure 9 for the resultant route/URL.

```
<a [routerLink]=”[‘/destination’]” fragment=”last” > <strong> Last >>> </strong>
</a>
```

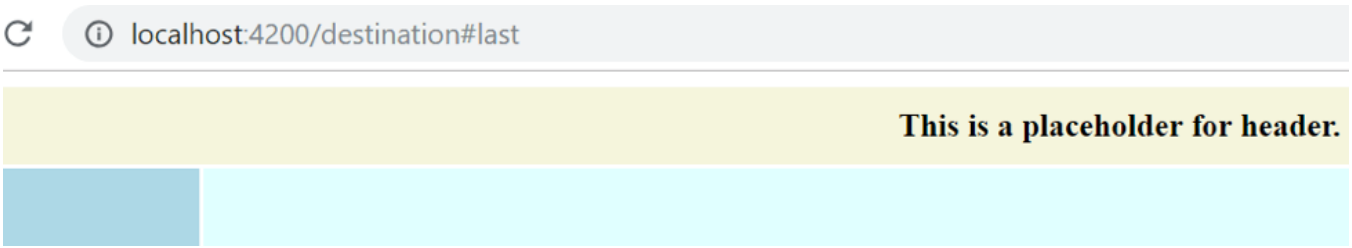


Figure 9: Fragment appended

With this, we conclude Part 1.

The purpose of Part 1 was to accustom you with the basics of routing. Part 1 began by introducing routing, details configuring routes and briefly described creating links. Angular monorepo has included routing package and the module.

Take a break, or go through the concepts again since we will be needing them in the second part of this article.

PART II – ANGULAR ROUTING – ADVANCED SCENARIOS

While the first part aimed as a getting started guide covering the basics of defining routes, Part 2 discusses some important features including asynchronously resolving a route after obtaining data, conditionally redirecting to an alternate route, authorization checks on a route etc.

Let us get started.

Base Element

This section describes a base element in index.html required for routing. It's always there for an application created with Angular-CLI. If your application is a custom setup, it's recommended to add this element.

A base element is required in `index.html`, with in the head tag. It defines how the relative path for the routing works. By default, it is set to a "/". We can change it to a different path indicating that the router will now use the new base root (for relative path).

With the base element shown here, the router will use /qa-instance-1 in the URL. This is useful when your application is not deployed at the root. Let's say, Index.html for the application maps to a domain-name.com/qa-instance-1. We can set the base URL so that all relative paths automatically use the base path.

```
<base href="/qa-instance-1">
```

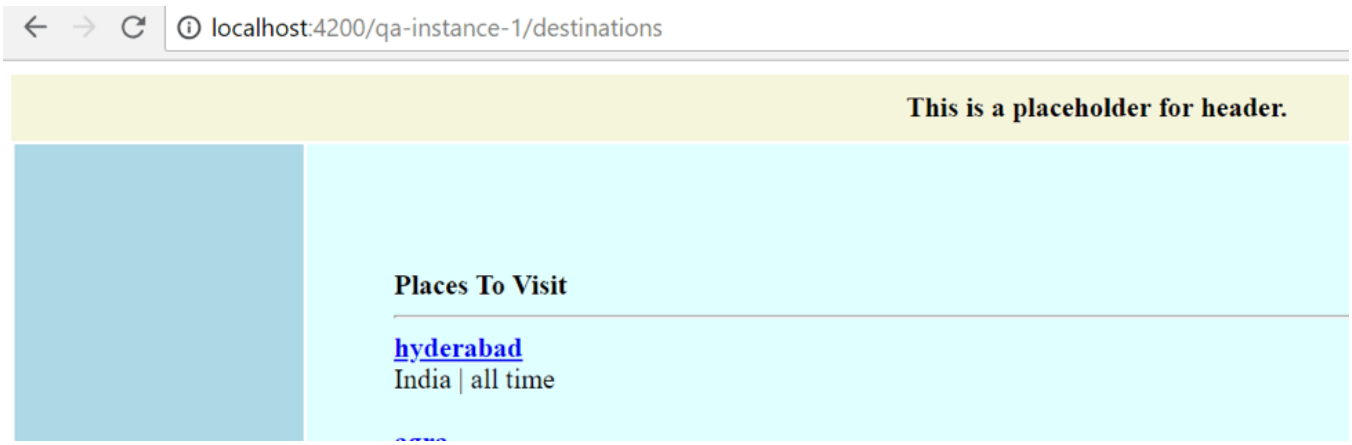


Figure 10: Base href set to qa-isntance-1

Please note, the base element could also have a `target` attribute. In an example, if target value is set to __blank, all hyperlinks open in a separate window. It doesn't affect the way Angular router works.

Location Strategy

This section compares retro style hash path in the URL with HTML5 style URL and routing. Angular router provides backward compatible hash path (#page1) in the route. By default, routes and URLs are seamless, for example my-domain-name.com/page1.

Yesteryear applications used URL in the link to decide,

1. Whether to reload the page or

2. Go to a section in the page without invoking server reload. The URL may have hash path/bookmark (eg. #section2) in which case there will not be a server reload. Without hash path if the URL changes, page will reload.

Modern browsers support navigating to a URL without a reload and a hash path. It would just be another URL, that doesn't reload from the server. In an example, navigation between `my-domain-name.com/page1` and `my-domain-name.com/page2` can happen without a server reload. There is no hash string required in the path.

Angular application provides both the options:

1. **PathLocationStrategy** – It is the default strategy with Angular router. In the code sample (My Travel), navigation between /destinations and /destination/city-name occurs without invoking a server reload.

2. **HashLocationStrategy** – The hash strategy is not recommended. If server-side rendering is required, this strategy doesn't integrate seamlessly. If you absolutely need this approach, provide the additional option `useHash` with a value `true` on the router module. Consider the following code snippet.

```
@NgModule({
  imports: [RouterModule.forRoot(routes, {useHash:true})],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

It prefixes every URL with a #. The URLs would be `my-domain-name/#/destinations` and `my-domain-name/#/destination/city-name`. See Figure-11.

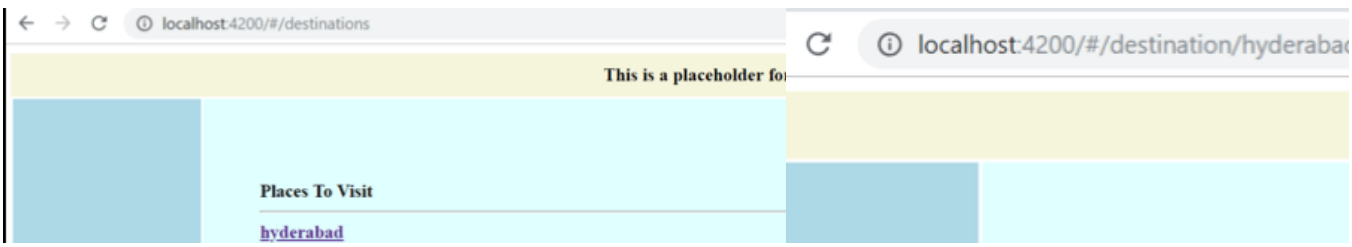


Figure 11: URL with HashLocationStrategy

More details

URL changes contained in the browser are made possible using the following HTML5 API - `pushState` and `replaceState` on `window.history` object. However, Angular Router by default uses `pushState` API.

Yesteryear browsers considered every new URL as a resource on the server. If we are moving from `my-domain.com/page1.html` to page2.html, the browser would request for page2.html which physically existed on the server. In case it's not a static html page, then ASP.NET or other similar server-side technologies created the content (or page2) on the fly.

`pushState` API allows changing the location on the window object without actually verifying the resource page2.html exists. `window.history` and state information are updated. It allows SPAs (single page applications) to navigate routes without server-reload.

Learn more about the HTML API at the following MDN Web Docs - https://developer.mozilla.org/en-US/docs/Web/API/History_API

Enable tracing of router events and data

We can enable tracing during navigation for debugging purposes. The Router events raised in the process are logged to the browser console.

Enable tracing by adding the flag while bootstrapping the router module. Consider following code snippet and Figure 12.

```
@NgModule({
  imports: [RouterModule.forRoot(routes, {enableTracing:true})],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

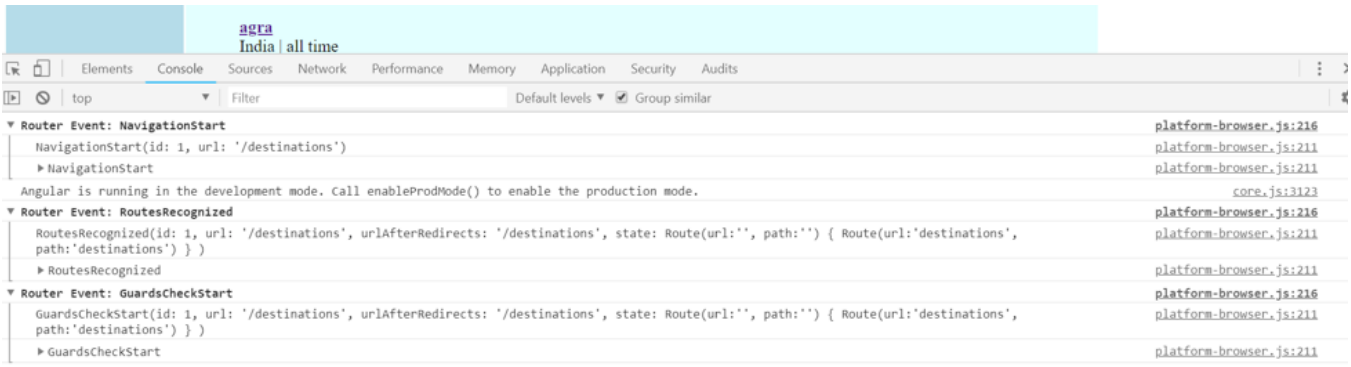


Figure 12: Router events captured with tracing

Router Events

Here's the complete list of Router Events. (reference: Angular documentation- https://angular.io/guide/router#router-events)

NavigationStart - This event occurs when navigation to the given route starts.
NavigationEnd - This event occurs when navigation to the given route ends.
NavigationCancel - An event occurs when navigation is canceled. This could be due to Route Guard or resolve returning false during navigation.
NavigationError - An event occurs when navigation fails with an exception.
RouteConfigLoadStart - This event occurs before route configuration is lazy loaded by the router.
RouteConfigLoadEnd - This event occurs after route configuration is lazy loaded by the router.
RoutesRecognized - This event occurs when router correctly parses and recognized route configuration.
GuardsCheckStart - Route Guards are used for authorization check. This event occurs at the beginning of

guard check.
GuardsCheckEnd - Route Guards are used for authorization check. This event occurs at the end of guard check.
ChildActivationStart - This even occurs before activating child route.
ChildActivationEnd - This even occurs after activating child route.
ActivationStart - This event occurs before activating/invoking a route.
ActivationEnd - This event occurs before activating/invoking a route.
ResolveStart - This event occurs at the beginning of resolving route configuration.
ResolveEnd - This event occurs at the end of resolving route configuration.

Please note, these events can be subscribed to and we can perform additional action when the event occurs. Consider the following code snippet in `ngOnInit` of a component. `router` is an instance of @ angular/router/Router.

```
this.router.events.subscribe(
  (event) =>
    console.log("Router event captured", event));
```

Here, `event` is an instance of any one of the above event classes. For example, it could be an instance of `NavigationStart`. All the above classes/events are inherited from `RouterEvents` class.

Static and Dynamic data to the component at route

This section describes passing static or dynamic data to the component at a given route. This data could be used in the component after the route is resolved. In a static data example, route configuration provides page title. In a dynamic data example, route configuration supplies an observable, resolving city details. Component just shows the provided city information.

Routing configuration can include additional data to be sent to the component. To demonstrate this functionality, consider the following example.

To the city details route, let's pass additional static data, *window title*. The Component can read this information and set the title.

```
const routes: Routes = [{
  path: "destination/:city",
  component: DestinationDetailsComponent,
  data: { title: "My City Details"}
  },…] // Routes configuration

@NgModule({
  imports: [RouterModule.forRoot(routes)], // set configured routes object
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

The Component has this data available at `this.route.snapshot.data`. We will use the `Title` service imported from '`@angular/platform-browser`'. Here's the code snippet.

```
constructor(private route: ActivatedRoute,
  private titleService: Title // Title Service injected
    …) { }
```

```
ngOnInit() {
  // Set title
  this.titleService.setTitle(this.route.snapshot.data.title || "");
```

Notice Figure 13 below. The Window title is set with static data sent from route configuration.

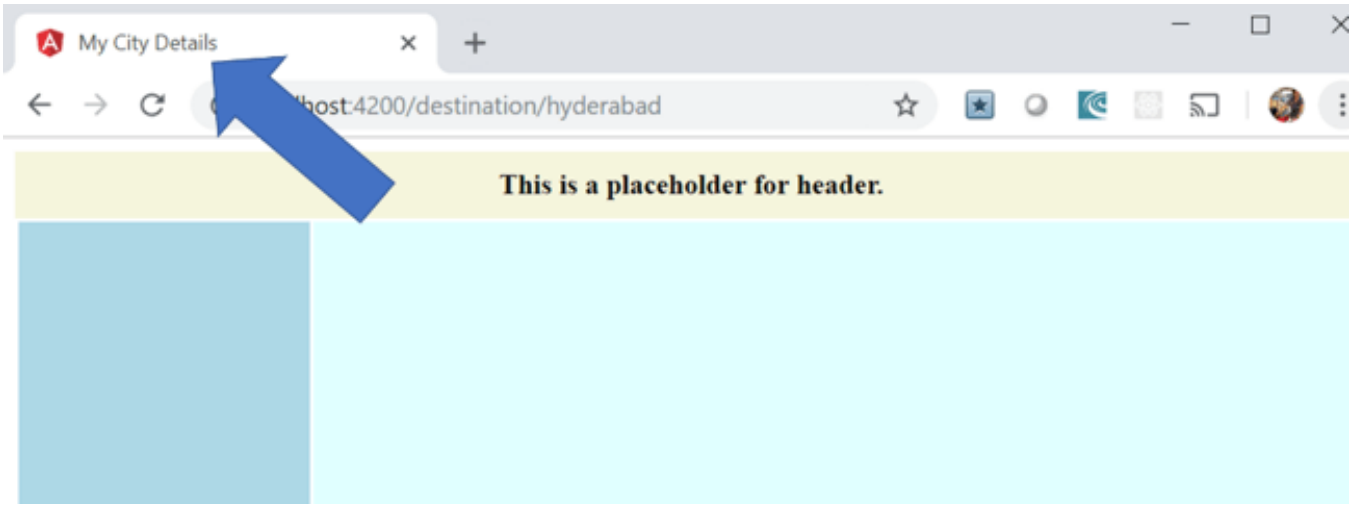


Figure 13: Window title set by the component

Fetch data on the fly before loading the route

While the data being passed to the route in the earlier example was static, we can obtain data on the fly.

This can be done asynchronously, via an observable or a promise, and will load the route only when resolved. (It may also return a dynamic value synchronously or instantly. If it's a time-consuming operation, preferably make it an observable or a promise).

We can write code to not redirect to the route when the data is not available. We may also redirect to a different route. Some prefer to fallback to an error page.

In the code sample we will see shortly, we will redirect to the *list screen* if a particular city data is unavailable.

1. As a first step we will create a new service that will fetch the data on the fly, before loading the route. If you are using Angular CLI, run the following command to create the new service.

   `ng generate service providers/destination-resolver`.

   This will create a DestinationResolverService in providers folder.

2. Consider implementing `Resolve` interface in @angular/router. While it's not mandatory, it's recommended. All that the Angular route configuration needs is the `resolve` function on the service. However, this interface ensures that the correct method signature is maintained. An incorrect function signature returns as an error at the build time.

Here's the code snippet.

```
import { Injectable } from '@angular/core';
```

```
import { Resolve, ActivatedRouteSnapshot, RouterStateSnapshot, Router } from '@
angular/router';
import { DataAccessService } from './data-access.service';
import { EMPTY, of } from 'rxjs';
import { take, mergeMap } from 'rxjs/operators';
import * as _ from 'lodash';

@Injectable({
  providedIn: 'root'
})
export class DestinationResolverService implements Resolve<any> {

 resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
  return this.dataAccess.queryDestinationByCity(route.params.city).pipe(
   take(1),
   mergeMap(city => {
    if (city && _.isArray(city) && city.length > 0) {
     return of(city); // returns observable based on the given object.
    } else { // if city details are not available, redirect to list screen
      this.router.navigate(['/destinations']);
      return EMPTY; // an empty observable
      }
   })
  );
 }

 constructor(private dataAccess: DataAccessService, private router: Router) { }
}
```

In the sample, the `ActivateRouteSnapshot` instance is used to get the selected city name from the URL (route parameter).

The `resolve` function queries city details with the help of `dataAccess` service and returns the resultant observable. Notice, if the city details are available, we return the resultant object, which could be used in the component. Otherwise, we redirect to */destinations* route, which is a list screen.

3. Use the resolver service (we just created) in the route configuration.

```
const routes: Routes = [
 {
  path: "destination/:city",
  component: PredefinedDestinationDetailsComponent,
  resolve: {
   destinationDetails: DestinationResolverService
  }
 },
// ... for brevity removing remaining route configuration objects
];

@NgModule({
 imports: [RouterModule.forRoot(routes)],
 exports: [RouterModule]
})
export class AppRoutingModule { }
```

Notice, the object passed to the component is referred to as `destinationDetails` as configured above.

4. Create a new component with Angular CLI to use the city details provided

```
ng generate component predefined-destination-details
```

5. The following code snippet reads dynamic data object obtained while resolving the route:

export class PredefinedDestinationDetailsComponent implements OnInit {

```
constructor(private router: ActivatedRoute) { }
 destination: any;

  ngOnInit() {
   this.router.data
    .subscribe( (data: {destinationDetails: any}) => {
      this.destination = data.destinationDetails[0];
  })
 }
}
```

This code sample obtains the object from the `ActivatedRoute` instance and sets it to a component level class variable. This object `destination` is used in the template file for showing city details. The code snippet below is for template details.

```
<div> <a routerLink="/destinations">Home</a> </div>
<strong>Welcome to {{destination && destination.name}}</strong>
<div>It's a city in {{destination && destination.country}}</div>
<div>Plan to visit in {{destination && destination.bestTime}}</div>
```

AUTH GUARD

This section describes authentication and authorization checks before redirecting to a route. Because they are asynchronous, they can be based on results from a server-side API call.

While the dynamic data with resolve retrieves data before navigating to the route, the same principle can be applied to authorization. Consider the following code snippet with `CanActivate`.

1. Create a new authorization service with Angular CLI

```
ng generate service authorization
```

2. Implement the `CanActivate` interface. The `canActivate` function will be invoked by router before activation. If the function returns `true` it will activate the route. When `false`, the service can redirect to an error page.

```
export class AuthorizationService implements CanActivate {

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
Observable<boolean>{
      // Perform authorization check
   // ...
      // End authorization check
       if(authorizationCheck){
```

```
            return of(true);
        }else{
this.router.navigate([‘/error’]);
return of(false);
        }
  }

  constructor(private router: Router ) { }
}
```

Note that the `canActivate` function can asynchronously return an `observable<boolean>` (like above code snippet), `promise<boolean>` or a synchronous Boolean value.

3. Configure the router to use an authorization check

```
 {
    path: “destinations”,
    component: DestinationListComponent,
    canActivate: [AuthorizationService]
  },
```

Here are additional configuration parameters for authorization check.

CanActivateChild: Authorization check to activate child routes. Use authorization service that implements `CanActivateChild` interface. When the `canActivateChild` function resolves to true, it will activate child routes.

CanDeactivateChild: Validation to discard changes at a route.

ActivatedRoute Vs ActivatedRouteSnapshot

In earlier examples, in a component at a route, when we wanted to read information from the URL or a route (be it route params, query string or fragment) we used `activatedRouteInstance.snapshot`.

Here, the snapshot is an object of `ActivatedRouteSnapshot`. It provides data in the URL at that point in time. It’s an immutable object, which means every time we read snapshot information with a different route param, a new object is created. Also, we shouldn’t inject `ActivateRouteSnapshot` as the immutable instance value doesn’t change.

There is another way to implement the same. Use observables on `ActivateRoute` instance, which change over time. Consider following code snippet

```
import { ActivatedRoute, ParamMap } from ‘@angular/router’;
import { switchMap } from ‘rxjs/operators’;
```

Import `ParamMap` for accessing route params. Import `switchMap` for iterating values returned by Observable.

In the snippet shown below, instead of using snaptshot on `this.route` (instance of ActivatedRoute) use `paramMap`, which returns an `Observable<ParamMap>`. Pipe the result with iterator, `switchMap`. Access the route parameter city in the `switchMap` implementation.

```
this.route.paramMap.pipe(
 switchMap((params: ParamMap) => this.dataAccess
  .getDestinationDetails(params.get('city')))
)
```

Conclusion

Routing is an important aspect of building a SPA (Single Page Application). All modern frameworks, be it Angular, React or Vue have a library for managing routing. Angular mono-repo comes with a routing module as well. It is well-rounded and provides all required features for taking care of navigation between views.

In this two-part tutorial on Angular Routing, we began with instructions to setup routing, then describe route configuration, an outlet where routing is applied and the links that are source of navigation.

In the second part, we got into the nitty-gritty details on features that control composing the URL (location strategy), additional optional configurations like the flag that enables tracing routing events. The tutorial then discussed how to conditionally control transition to a route based on an API response, be it the data for the component or authentication/authorization check.

References:

Angular documentation on routing- https://angular.io/guide/router

MDN Web Docs - https://developer.mozilla.org/en-US/docs/Web/API/History_API

Blog: Understanding Router State- https://vsavkin.com/angular-router-understanding-router-state-7b5b95a12eab

RouterEvent documentation - https://angular.io/api/router/RouterEvent

• • • • • • •

Download the entire source code from GitHub at
bit.ly/dncm39-angular-routing

Microsoft MVP Most Valuable Professional

Keerti Kotaru
Author

V Keerti Kotaru has been working on web applications for over 15 years now. He started his career as an ASP.Net, C# developer. Recently, he has been designing and developing web and mobile apps using JavaScript technologies. Keerti is also a Microsoft MVP, author of a book titled 'Material Design Implementation using AngularJS' and one of the organisers for vibrant ngHyderabad (AngularJS Hyderabad) Meetup group. His developer community activities involve speaking for CSI, GDG and ngHyderabad.

Thanks to Ravi Kiran *for reviewing this article.*

Damir Arh

HOW TO CHOOSE THE RIGHT .NET COLLECTION CLASS?

The .NET framework Base Class Library contains many collection classes. This can make it difficult to decide when to use which. Grouping them together based on their properties can make the choice for a specific scenario, much easier.

Generic Collections

Most of the collection classes are placed in the `System.Collections` and the `System.Collections.Generic` namespaces, but many of them are placed in other namespaces as well. Although, in most cases, only collections in the `System.Collections.Generic` namespace should be considered.

The collections in `System.Collections` are all non-generic, originating from the time before generics were added to C# with the release of .NET framework 2.0 and C# 2.0. Unless you are maintaining legacy code, you should avoid them because these don't provide type safety and *have the worst performance* when compared to their generic counterparts when used with value types.

Editorial Note: *Read Using Generics in C# to Improve Application Maintainability to make our software more resilient to data-related changes, thereby improving its maintainability.*

The collections in other namespaces are highly specialized for specific scenarios and are usually not applicable to code from other problem domains. E.g. the `System.Dynamic.ExpandoObject` class implements multiple collection interfaces but is primarily meant for implementing dynamic binding. Even when only looking at generic collections in the `System.Collections.Generic` namespace, the choice can still seem overwhelming.

Fortunately, the collections can be easily grouped in a few categories based on the interfaces they implement. These determine which operations are supported by a collection and consequently in which scenarios can they be used.

The common interface for collections is the `ICollection<T>` interface. It inherits from the `IEnumerable<T>` interface which provides the means for iterating through a collection of items:

```
IEnumerable<int> enumerable = new List<int>() { 1, 2, 3, 4, 5 };

foreach(var item in enumerable)
{
  // process items
}
```

The `ICollection<T>` interface adds the Count property and methods for modifying the collection:

```
ICollection<int> collection = new List<int>() { 1, 2, 3, 4, 5 };

var count = collection.Count;
collection.Add(6);
collection.Remove(2);
collection.Clear();
```

The authors of the Base Class Library (BCL) believed that these suffice for implementing a simple collection. Three different interfaces extend this base interface in different ways to provide additional functionalities.

1. Lists

The `IList<T>` interface describes collections with items which can be accessed by their index. For this purpose, it adds an indexer for getting and setting a value:

```
IList<int> list = new List<int> { 1, 2, 3, 4, 5 };

var item = list[2]; // item = 3
list[2] = 6;        // list = { 1, 2, 6, 4, 5 }
```

It also includes methods for inserting and removing a value at a specific index:

```
list.Insert(2, 0); // list = { 1, 2, 0, 6, 4, 5 }
list.RemoveAt(2);  // list = { 1, 2, 6, 4, 5 }
```

The most commonly used class implementing the `IList<T>` interface is `List<T>`. It will work great in most scenarios, but there are other implementations available for more specific requirements. For example, the `SynchronizedCollection<T>` has a built-in synchronization object which can be used to make it thread-safe.

2. Sets

`ISet<T>` interface describes a set, i.e. a collection of unique items which doesn't guarantee to preserve their order. The `Add` method can still be used for adding individual items to the collection:

```
ISet<int> set = new HashSet<int> { 1, 2, 3, 4, 5 };

var added = set.Add(0);
```

However, it will only add the item to the collection if it's not already present in it. The return value will indicate if the item was added.

Rest of the methods in the interface implement different standard set operations from set theory. The following methods modify the existing collection:

```
ISet<int> set = new HashSet<int> { 1, 2, 3, 4, 5 };
set.UnionWith(new[] { 5, 6 });              // set = { 1, 2, 3, 4, 5, 6 }
set.IntersectWith(new[] { 3, 4, 5, 6, 7 }); // set = { 3, 4, 5, 6 }
set.ExceptWith(new[] { 6, 7 });             // set = { 3, 4, 5 }
set.SymmetricExceptWith(new[] { 4, 5, 6 }); // set = { 3, 6 }
```

The others only perform tests on the collection and return a `boolean` value:

```
ISet<int> set = new HashSet<int> { 1, 2, 3, 4, 5 };
var isSubset = set.IsSubsetOf(new[] { 1, 2, 3, 4, 5 }); // = true
var isProperSubset = set.IsProperSubsetOf(new[] { 1, 2, 3, 4, 5 }); // = false
var isSuperset = set.IsSupersetOf(new[] { 1, 2, 3, 4, 5 }); // = true
var isProperSuperset = set.IsProperSupersetOf(new[] { 1, 2, 3, 4, 5 }); // = false
var equals = set.SetEquals(new[] { 1, 2, 3, 4, 5 }); // = true
var overlaps = set.Overlaps(new[] { 5, 6 }); // = true
```

The most basic implementation of `ISet<T>` is the `HashSet<T>` class. If you want the items in the set to be sorted, you can use `SortedSet<T>` instead.

3. Dictionaries

`IDictionary<TKey, TValue>` is the third interface extending the `ICollection<T>` interface. In contrast to the previous two, this one is for storing key-value pairs instead of standalone values, i.e. it uses

`KeyValuePair<TKey, TValue>` as the generic parameter in `ICollection<T>`. Its members are designed accordingly. The indexer allows getting and setting the values based on a key instead of an index:

```
IDictionary<string, string> dictionary = new Dictionary<string, string>
{
  [“one”] = “ena”,
  [“two”] = “dva”,
  [“three”] = “tri”,
  [“four”] = “štiri”,
  [“five”] = “pet”
};
var value = dictionary[“two”];
dictionary[“zero”] = “nič” ;
```

The `Add` method can be used instead of the indexer to add a pair to the collection. Unlike the indexer it will throw an exception if the key is already in the collection.

```
dictionary.Add(“zero”, “nič”);
```

Of course, there are methods for checking if a key is in the collection and for removing an existing pair based on its key value:

```
var contains = dictionary.ContainsKey(“two”);
var removed = dictionary.Remove(“two”);
```

The latter will return a value indicating whether or not the key was removed because it wasn’t there in the collection in the first place.

There are also properties available for retrieving separate collections of keys and values:

```
ICollection<int> keys = dictionary.Keys;
ICollection<string> values = dictionary.Values;
```

For scenarios with no special requirements, the `Dictionary<TKey, TValue>` class is the go-to implementation of the `IDictionary<TKey, TValue>` interface.

Two different implementations are available if keys need to be sorted (`SortedDictionary<TKey, TValue>` and `SortedList<TKey, TValue>`), each with its own advantages and disadvantages.

Many more implementations of the interface are available in the Base Class Library.

Queue and Stack

There are two collection classes worth mentioning which unlike the others so far, implement none of the specialized interface derived from the `ICollection<T>` interface.

The `Queue<T>` class implements a FIFO (first in, first out) collection.

Only a single item in it is directly accessible, i.e. the one that’s in it for the longest time. The methods for adding and removing an item have standard names for such a datatype:

```
var queue = new Queue<int>(new[] { 1, 2, 3, 4, 5 });
queue.Enqueue(6);
```

```
var dequeuedItem = queue.Dequeue();
```

There's also the `Peek` method which returns the same item as the `Dequeue` method but leaves it in the collection:

```
var peekedItem = queue.Peek();
```

The `Stack<T>` class is similar to `Queue<T>`, but it implements a LIFO (last in, first out) collection. The single item that's directly accessible in this collection is the one that was added the most recently. The methods names reflect this:

```
var stack = new Stack<int>(new[] { 1, 2, 3, 4, 5 });
stack.Push(6);
var poppedItem = stack.Pop();
var peekedItem = stack.Peek();
```

Thread Safety

The regular generic classes in the Base Class Library have one very important deficiency if you need to use them in a multi-threaded application: **they are not thread-safe**.

...or at least not entirely thread-safe.

While most of them support several concurrent readers, the reading operations are still not thread-safe as no concurrent write access is allowed.

Editorial Note: *Read* C# Sharding and Multithreading - Deep Dive *as well as* Concurrent Programming in .NET Core.

As soon as the collection has to be modified, any access to it from multiple threads must be synchronized.

The simplest approach to implementing such synchronization involves using the lock statement with a common synchronization object:

```
public class DictionaryWithLock<TKey, TValue>
{
  private readonly object syncObject = new object();
  private readonly Dictionary<TKey, TValue> dictionary;

  public DictionaryWithLock()
  {
    dictionary = new Dictionary<TKey, TValue>();
  }

  public TValue this[TKey key]
  {
    get
    {
      lock (syncObject)
      {
         return dictionary[key];
      }
    }
    set
```

```
    {
      lock (syncObject)
      {
        dictionary[key] = value;
      }
    }
  }

  public void Add(TKey key, TValue value)
  {
    lock (syncObject)
    {
      dictionary.Add(key, value);
    }
  }
}
```

Although this approach would work, it is far from optimal.

All access to the inner dictionary is fully serialized, i.e. the next operation can only start when the previous one is completed. Since the collection allows multiple concurrent readers, it can significantly improve performance if it takes that into account and only restricts concurrent access for writing operations.

Fortunately, there's no need for implementing this functionality from scratch.

The Base Class Library comes with the `ReaderWriterLockSlim` class which can be used to implement this specific functionality in a relatively simple manner:

```
public class DictionaryWithReaderWriterLock<TKey, TValue>
{
  private readonly ReaderWriterLockSlim dictionaryLock = new
  ReaderWriterLockSlim();
  private readonly Dictionary<TKey, TValue> dictionary;

  public DictionaryWithReaderWriterLock()
  {
    dictionary = new Dictionary<TKey, TValue>();
  }

  public TValue this[TKey key]
  {
    get
    {
      dictionaryLock.EnterReadLock();
      try
      {
        return dictionary[key];
      }
      finally
      {
        dictionaryLock.ExitReadLock();
      }
    }
    set
    {
      dictionaryLock.EnterWriteLock();
      try
      {
```

```
        dictionary[key] = value;
      }
      finally
      {
        dictionaryLock.ExitWriteLock();
      }
    }
  }

  public void Add(TKey key, TValue value)
  {
    dictionaryLock.EnterWriteLock();
    try
    {
      dictionary.Add(key, value);
    }
    finally
    {
      dictionaryLock.ExitWriteLock();
    }
  }
}
```

There are two different lock methods used, depending on the type of operation being synchronized: reading or writing.

The class will allow any number of concurrent read operations. On the other hand, the write lock will be exclusive and won't allow any other read or write access at the same time.

Concurrent Collections

Even with the use of synchronization primitives provided by C# and the .NET framework, writing production-quality synchronization code is no small feat. Both of my wrapper classes implement synchronization for only a small subset of operations. You would want to implement at least the full `IDictionary<TKey, TValue>` interface if not all the members of the wrapped `Dictionary<TKey, TValue>` class.

That's exactly what the concurrent collections in the `System.Collections.Concurrent` namespace do. They provide thread-safe implementations of collection interfaces.

The exceptions are the `ConcurrentQueue<T>` and the `ConcurrentStack<T>` classes which provide independent implementations similar to their non-concurrent counterparts `Queue<T>` and `Stack<T>` because there are no suitable interfaces in the .NET framework to implement.

The concurrent collection classes can be safely used in multi-threaded applications. They even have extra members in addition to those from the implemented interfaces which can prove useful in multi-threaded scenarios.

Still, there are minor caveats when using these collections. For example, the `ConcurrentDictionary<TKey, TValue>` class includes methods which are not fully atomic. The overloads of `GetOrAdd` and `AddOrUpdate` methods which accept delegates as parameters will invoke these delegates outside the synchronization locks.

Let's inspect the following line of code to understand the implications of this:

```
var resource = concurrentDictionary.GetOrAdd(newKey, key => ExpensiveResource.
Create(key));
```

Such method calls are common when the dictionary is used as a cache for instances of classes which might be expensive to create but can safely be used from multiple threads.

Without knowing the details of how the `ConcurrentDictionary<TKey, TValue>` is implemented, one would assume that this line of code will either return an instance from the cache or create a single new instance using the provided delegate when there's no matching instance yet in the dictionary. It should then return that instance on all subsequent requests.

However, since the delegate is invoked *outside* the synchronization lock, it could be invoked from multiple different threads if it is not yet in the dictionary when this line of code is reached.

Although only one of the created instances will be stored in the dictionary in the end, the fact that multiple instances were created could be an issue. At the very least, it will affect performance if the creation takes a long time. If the factory method should only ever be called once for a specific key, you will need to write additional synchronization code yourself.

It is important to always thoroughly read the documentation for all concurrent collection classes to be aware of such implementation details.

Immutable Collections

Immutable collections aren't included in the Base Class Library. To use them, the `System.Collections.Immutable` NuGet package must be installed in the project.

They take a different approach to making collections thread-safe. Instead of using synchronization locks as concurrent collections do, immutable collections can't be changed after they are created. This automatically makes them safe to use in multi-threaded scenarios since there's no way for another thread to modify them and make the state inconsistent.

This fundamental design decision affects the API of immutable collection classes. They don't even have public constructors. There are two other ways to create a new instance though:

- A regular collection can be converted into an immutable one using an extension method:

```
var immutableList = new[] { 1, 2, 3, 4, 5 }.ToImmutableList();
```

- An empty collection which is exposed as a static property can be modified by adding new items to it:

```
var immutableList = ImmutableList<int>.Empty.AddRange(new[] { 1, 2, 3, 4, 5 });
```

All operations which would normally modify the collection return a new instance instead. It's important to remember that the returned value must be used from there on instead of the original instance:
`immutableList = immutableList.Add(6);`
That each method returns a new instance of the same class makes it easy to chain multiple method calls:

```
immutableList = immutableList
  .Add(6)
```

```
  .Add(7)
  .Add(8);
```

However, this approach should be avoided whenever possible because each method call in such a chain creates a new instance of the collection. Although the immutable collections are implemented in a way to reuse as much of the original collection as possible when creating a new instance, some memory allocations are still required. This means more work for the garbage collector.

The best way to minimize this problem is to use methods which can perform the required modifications in a single call. The above chain of method calls could for example be replaced with the following single method call:

```
immutableList = immutableList.AddRange(new[] { 6, 7, 8 });
```

Even more complex operations can be performed with a single method call. By following the naive approach from mutable collections, the following block of code would be used to remove all even numbers from a list:

```
for (var i = immutableList.Count - 1; i >= 0; i--)
{
  if (immutableList[i] % 2 == 0)
  {
    immutableList = immutableList.RemoveAt(i);
  }
}
```

To improve performance, the following equivalent line of code should be used instead:

```
immutableList = immutableList.RemoveAll(n => n % 2 == 0);
```

However, specialized methods are not available for all types of complex modifications. For example, there's no single method available to both add and remove specific items as below:

```
immutableList = immutableList
  .Add(6)
  .Remove(2);
```

In such cases a builder can be used instead:

```
var builder = immutableList.ToBuilder();
builder.Add(6);
builder.Remove(2);
immutableList = builder.ToImmutable();
```

The `ToBuilder` method will create a builder for an immutable collection which will implement the interface of the corresponding mutable collection. Its internal memory structure will still match the one of the immutable collection, but the operations will modify the same instance instead of always creating a new one. Only when calling the `ToImmutable` method, will the instance be made immutable again very efficiently. This will reduce the amount of work for the garbage collector as much as possible.
So, when should immutable collections be used instead of regular or concurrent mutable ones?

A typical use case are multi-threaded applications, especially when a thread need not have access to the latest state of the collection and can use a potentially stale immutable instance which was originally

passed to it. When that's the case, immutable collections might offer better performance despite the additional work for the garbage collector because there are no synchronization locks required.

If an application needs to undo the operations on collections, it might make sense to use immutable collections even in single-threaded scenarios. Snapshots of previous states are a side product of immutable collections and require none additional resources to create. They can, for example, be stored in a stack to undo the last change:

```
snapshots.Push(immutableList);
immutableList = immutableList.Add(6);

immutableList = snapshots.Pop(); // undo: revert list to previous state
```

To achieve the same functionality with mutable collections, copies must be created before each modification:

```
snapshots.Push(list.ToList());
list.Add(6);

list = snapshots.Pop();
```

Not only is creating copies time consuming, the copies also require more memory than their immutable counterparts. The copies of mutable collections don't share any memory between them, unlike immutable collections which often share large parts when they were created through modifications.

Conclusion:

When choosing a collection to use in your code, always start by thinking through which operations you will need to perform on that collection. Based on that, you can select the most appropriate collection interface.

Unless you have any other special requirements, go with an implementation from the `System.Collections.Generic namespace`. If you're writing a multithreaded application and will need to modify the collection from multiple threads, choose the concurrent implementation of the same interface instead. Consider immutable collections if their behavior and performance match your requirements best.

Microsoft® MVP Most Valuable Professional

Damir Arh

Author

Damir Arh has many years of experience with Microsoft development tools; both in complex enterprise software projects and modern cross-platform mobile applications. In his drive towards better development processes, he is a proponent of test driven development, continuous integration and continuous deployment. He shares his knowledge by speaking at local user groups and conferences, blogging, and answering questions on Stack Overflow. He is an awarded Microsoft MVP for .NET since 2012.

Thanks to Yacoub Massad *for reviewing this article.*

THANK YOU

FOR THE 39th EDITION


@yacoubmassad


@Rahul1000Buddhe

@damirarh

@sravi_kiran


@sandeepchads


@WestDiscGolf

@keertikotaru

@suprotimagarwal

@techieshravan

@subodhsohoni


@saffronstroke

WRITE FOR US

mailto: suprotimagarwal@dotnetcurry.com