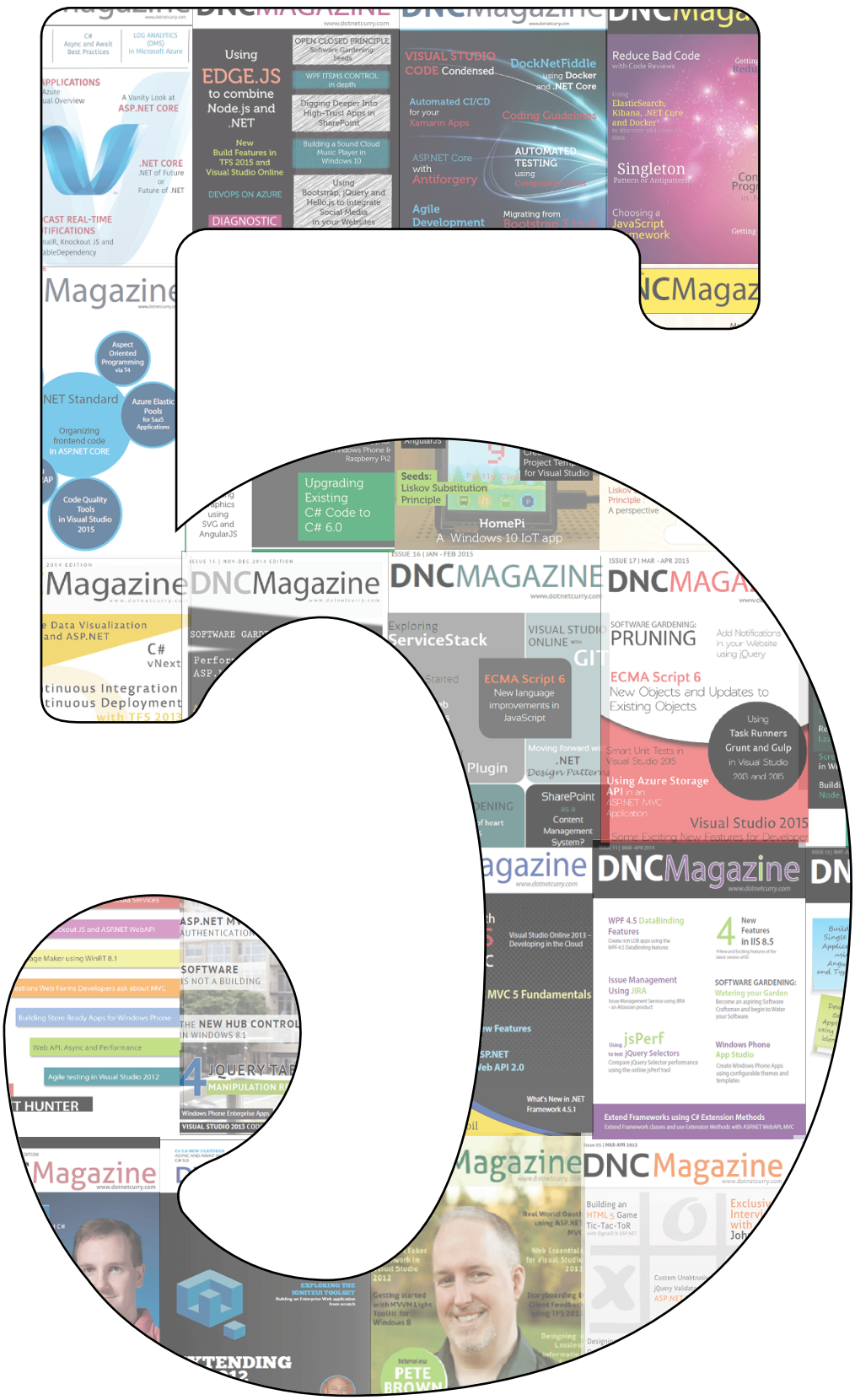
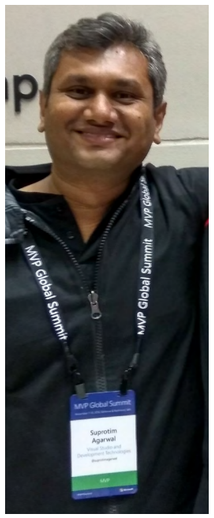
Issue 31 | July-Aug 2017

DNCMagazine

www.dotnetcurry.com



FROM THE EDITOR



How time flies! ***DNC Magazine,*** *the digital publication dedicated to .NET and JavaScript developers, is already five years old. 31 issues, millions of downloads, hundreds of useful articles later, there's barely any time to take a breather as we try and keep up with the ever changing developer landscape. This milestone not only marks 5 years of the DNC Magazine, it's also a springboard, as we continue to educate and empower developers to boost their career.*

It is impossible to thank anyone enough for all the support they have extended over this time. Nevertheless I want to take a moment and thank our extraordinary team of MVPs and experts who I am humbled to work with. I also want to extend by heartfelt thanks to our sponsors Arction, Docuvieware, RavenDB, Recime and others.

And to you, our readers; for without you, we wouldn't exist. Thank you!

5 years down, foreover to go!!

Suprotim Agarwal

Editor in Chief

suprotimagarwal@dotnetcurry.com

THE TEAM

Editor In Chief
Suprotim Agarwal

Art Director
Minal Agarwal

Contributing Authors
Damir Arh
Daniel Jimenez Garcia
Gerald Versluis
Keerti Kotaru
Mehfuz Hossain
Rahul Sahasrabuddhe
Ravi Kiran
Vikram Pendse
Yacoub Massad

Technical Reviewers
Damir Arh
Kunal Chandratre
Rahul Sahasrabuddhe
Ravi Kiran
Suprotim Agarwal
Vikram Pendse
Yacoub Massad

Next Edition
September 2017

www.dotnetcurry.com/magazine

THANK YOU

FOR THE 31st EDITION


@damirarh


@yacoubmassad


@ravikiran


@dani_djg


@vikrampendse


@rahul1000buddhe


@keertikotaru


@jfversluis


Sanjeev Assudani


@suprotimagarwal


@kunalchandratre


@ saffronstroke

WRITE FOR US

mailto: suprotimagarwal@dotnetcurry.com

Why **Fortune 500** companies choose **RavenDB**?

In a world where data is one of the most important assets of any business the database technology should not only be protecting its data but also enhancing its business.

To address both of those needs, Hibernating Rhinos has introduced its NoSQL database called RavenDB and for the past few years, due to enhanced capabilities, it has become the choice of Fortune 500 companies.

The protection of data comes with meeting all the ACID parameters, being fully transactional and having extended failover support to guarantee you that the data will be safe and sound even when node failure happens. Moreover, the extended replication features allow businesses to setup complex failover clusters to move their protection to the next level and ensure availability or enhance their work by enabling sophisticated sharding and load balancing capabilities.

The out of the box querying features, high-performance and self-optimization assure that the database will not stand in the way of company growth.

All this is provided with user-friendly HTML5 management interface, ease of deployment and top-notch C# and Java client libraries.

- **Schema-free**
- **Scalable**
- **RavenFS**
- **Easy to use**
- **Transactional**
- **High Performance**
- **Extensible**
- **Designed with Care**

NEW

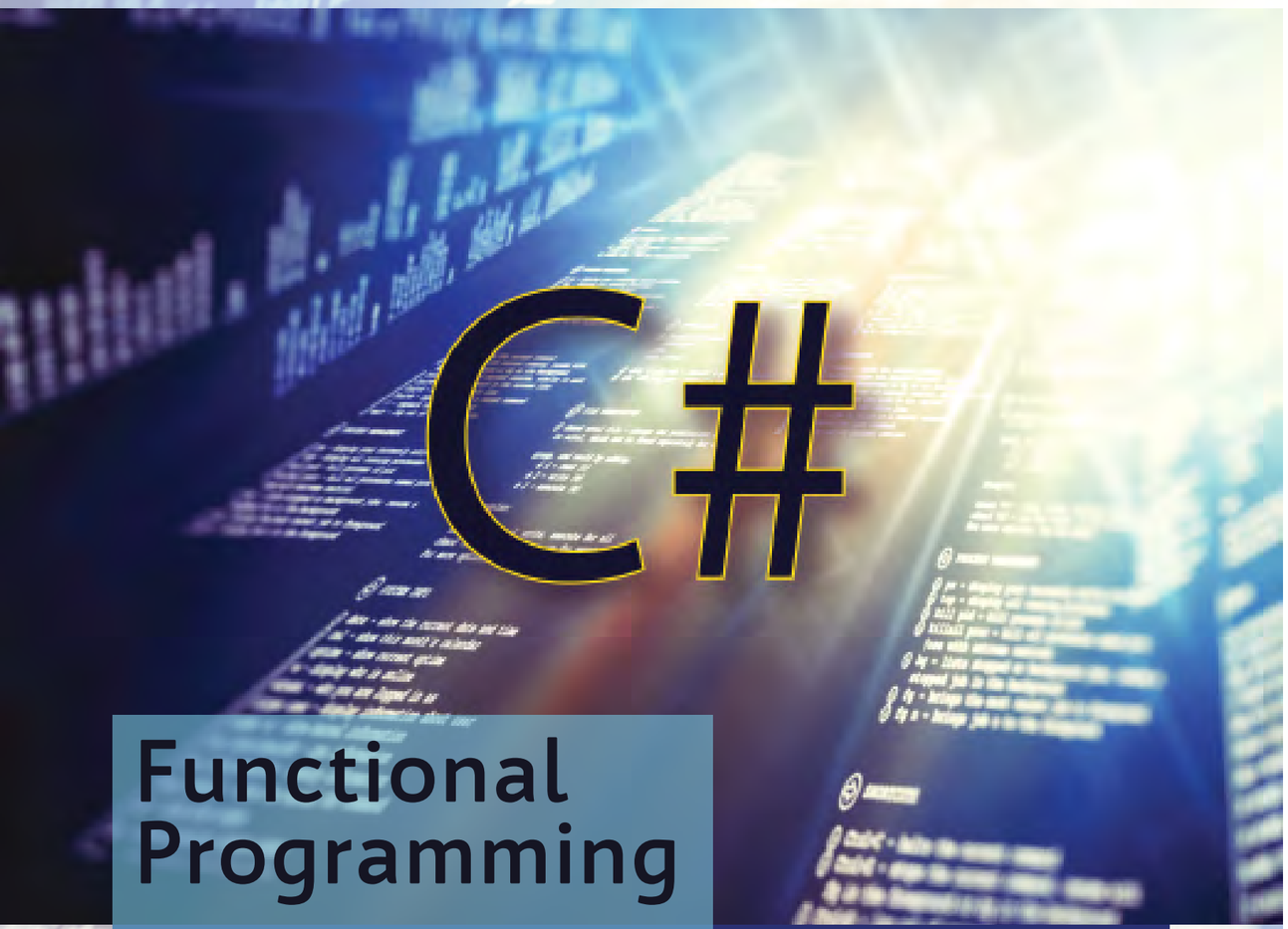
- **Monitoring**
- **Hot Spare**
- **Clustering**

RAVENDB 3.5 RELEASED

ravendb.net

THE CONTENTS

Damir Arh



Functional Programming

for C# Developers

Functional programming seems to be gaining popularity recently. The functional programming language for .NET framework is F#. However, although C# is an object oriented language at its core, it also has a lot of features that can be used with functional programming techniques. You might already be writing some functional code without realizing it.

Functional Programming Paradigm

Functional programming is an alternative programming paradigm to the currently more popular and common, object-oriented programming.

There are several key concepts that differentiate it from the other programming paradigms. Let's start by providing definitions for the most common ones, so that we will recognize them when we see them applied throughout the article.

The basic building blocks of functional programs are **pure functions**. They are defined by the following two properties:

- Their result depends solely on the arguments passed to it. No internal or external state affects it.
- They do not cause any side effects. The number of times they are called will not change the program behavior.

Because of these properties, a function call can be safely replaced with its result, e.g. to cache the results of computationally intensive functions for each combination of its arguments (technique known as memoization).

Pure functions lend themselves well to **function composition**.

This is a process of combining two or more functions into a new function, which returns the same result as if all its composing functions were called in a sequence. If `ComposedFn` is a function composition of `Fn1` and `Fn2`, then the following assertion will always pass:

```
Assert.That(ComposedFn(x), Is.EqualTo(Fn2(Fn1(x))));
```

Composition is an important part of making functions reusable.

Having functions as arguments to other functions can further increase their reusability. Such **higher-order functions** can act as generic helpers, which apply another function passed as argument multiple times, e.g. on all items of an array:

```
Array.Exists(persons, IsMinor);
```

In the above code, `IsMinor` is a function, defined elsewhere. For this to work, the language must support **first-class functions**, i.e. allow functions to be used as first-class language constructs just like value literals.

Data is always represented with **immutable objects**, i.e. objects that cannot change their state after they have been initially created. Whenever a value changes, a new object must be created instead of modifying the existing one. Because all objects are guaranteed to not change, they are inherently thread-safe, i.e. they can be safely used in multithreaded programs with no threat of race conditions.

As a direct consequence of functions being pure and objects being immutable, there is **no shared state** in functional programs.

Functions can act only based on their arguments, which they cannot change and therewith, affect other functions receiving the same arguments. The only way they can affect the rest of the program is through

the result they return, which will be passed on as arguments to other functions.

This prevents any kind of hidden cross-interaction between the functions, making them safe to run in any order or even in parallel, unless one function directly depends on the result of the other.

With these basic building blocks, functional programs end up being more declarative than imperative, i.e. instead of describing *how* to calculate the result, the programmer rather describes *what* to calculate.

The following two functions that convert the *case* of an array of strings to lower case, clearly demonstrate the difference between the two approaches:

```
string[] Imperative(string[] words) {
  var lowerCaseWords = new string[words.Length];
  for (int i = 0; i < words.Length; i++)
  {
    lowerCaseWords[i] = words[i].ToLower();
  }
  return lowerCaseWords;
}

string[] Declarative(string[] words)
{
  return words.Select(word => word.ToLower()).ToArray();
}
```

Although you will hear about many other functional concepts, such as monads, functors, currying, referential transparency and others, these building blocks should suffice to give you a basic idea of what functional programming is and how it differs from object-oriented programming.

Writing Functional Code in C#

You can implement many functional programming concepts in C#.

Since the language is primarily object-oriented, the defaults don't always guide you towards such code, but with intent and enough self-discipline, your code can become much more functional.

Immutable Types

You most probably are used to writing mutable types in C#, but with very little effort, they can be made immutable:

```
public class Person
{
  public string FirstName { get; private set; }
  public string LastName { get; private set; }
  public Person(string firstName, string lastName)
  {
    FirstName = firstName;
    LastName = lastName;
  }
}
```

Private property setters make it impossible to assign them a different value after the object has been initially created. For the object to be truly immutable, all the properties must also be of immutable types. Otherwise their values can be changed by mutating the properties, instead of assigning a new value to them.

The `Person` type above is immutable, because `string` is also an immutable type, i.e. its value cannot be changed as all its instance methods return a new `string` instance. However this is an exception to the rule and most .NET framework classes are mutable.

If you want your type to be immutable, you should not use any other built-in type other than primitive types, and strings as public properties.

To change a property of the object, e.g. to change the person's first name, a new object needs to be created:

```
public static Person Rename(Person person, string firstName)
{
  return new Person(firstName, person.LastName);
}
```

When a type has many properties, writing such functions can become quite tedious. Therefore, it is a good practice for immutable types to implement a `With` helper function for such scenarios:

```
public Person With(string firstName = null, string lastName = null)
{
  return new Person(firstName ?? this.FirstName, lastName ?? this.LastName);
}
```

This function creates a copy of the object with any number of properties modified. Our `Rename` function can now simply call this helper to create the modified person:

```
public static Person Rename(Person person, string firstName)
{
  return person.With(firstName: firstName);
}
```

The advantages might not be obvious with only two properties, but no matter how many properties the type consists of, this syntax allows us to only list the properties we want to modify as *named* arguments.

Pure Functions

Making functions 'pure' requires even more discipline than making objects immutable.

There are no language features available to help the programmer ensure that a particular function is pure. It is your own responsibility to not use any kind of internal or external state, to not cause side effects and to not call any other functions that are not pure.

Of course, there is also nothing stopping you from only using the function arguments and calling other pure functions, thus making the function pure. The Rename function above is an example of a pure function: it does not call any non-pure functions or use any other data than the arguments passed to it.

Function Composition

Multiple functions can be composed into one by defining a new function, which calls all the composed functions in its body (let us ignore the fact that there is no need to ever call Rename multiple times in a row):

```
public static Person MultiRename(Person person) {
  return Rename(Rename(person, “Jane”), “Jack”);
}
```

The signature of `Rename` method forces us to nest the calls, which can become difficult to read and comprehend, as the number of function calls increases. If we use the `With` method instead, our intent becomes clearer:

```
public static Person MultiRename(Person person) {
  return person.With(firstName: “Jane”).With(firstName: “Jack”);
}
```

To make the code even more readable, we can break the chain of calls into multiple lines, keeping it manageable, no matter how many functions we compose into one:

```
public static Person MultiRename(Person person)
{
  return person
    .With(firstName: “Jane”)
    .With(firstName: “Jack”);
}
```

There is no good way to split lines with `Rename`-like nested calls. Of course, `With` method allows the chaining syntax due to the fact that it is an instance method.

However, in functional programming, functions should be declared separately from the data they act upon, like Rename function is. While functional languages have a pipeline operator (|> in F#) to allow chaining of such functions, we can take advantage of extension methods in C# instead:

```
public static class PersonExtensions

{
  public static Person Rename(this Person person, string firstName)
  {
    return person.With(firstName: firstName);
  }
}
```

This allows us to chain non-instance method calls, the same way as we can instance method calls:

```
public static Person MultiRename(Person person)
{
  return person.Rename(“Jane”).Rename(“Jack”);
}
```

Examples of Functional APIs in .NET Framework

To have a taste of functional programming in C#, you don't need to write all the objects and functions yourself. There are some readily available functional APIs in .NET framework for you to utilize.

Immutable Collections

We have already mentioned `string` and primitive types as immutable types in .NET framework.

However, there is also a selection of immutable *collection* types available. Technically, they are not really a part of the .NET framework, since they are distributed out-of-band as a stand-alone NuGet package **System.Collections.Immutable**.

On the other hand, they are an integral part of .NET Core, the new open-source cross-platform .NET runtime.

The namespace includes all the commonly used collection types: array, lists, sets, dictionaries, queue and stack. As the name implies, all of them are immutable, i.e. they cannot be changed after they are created. Instead a new instance is created for every change. This makes the immutable collections completely thread-safe in a different way than the concurrent collections, which are also included in the .NET framework base class library.

With concurrent collections, multiple threads cannot modify the data simultaneously but they still have access to the modifications. With immutable collections, any changes are only visible to the thread that made them, as the original collection remains unmodified.

To keep the collections performant in spite of creating a new instance for every mutable operation, their implementation takes advantage of **structural sharing**. This means that in the new modified instance of the collection, the unmodified parts from the previous instance are reused as much as possible, thus requiring less memory allocation and causing less work for the garbage collector.

This common technique in functional programming is made possible by the fact that objects cannot change and can therefore be safely reused.

The biggest difference between using immutable collections and regular collections, is in their creation. Since a new instance is created on every change, you want to create the collection with all the initial items already in it. As a result, immutable collections don't have public constructors, but offer three alternative ways of creating them:

- Factory method `Create` accepts 0 or more items to initialize the collection with:

```
var list = ImmutableList.Create(1, 2, 3, 4);
```

- `Builder` is an efficient mutable collection that can be easily converted to its immutable counterpart:

```
var builder = ImmutableList.CreateBuilder<int>();
builder.Add(1);
builder.AddRange(new[] { 2, 3, 4 });
var list = builder.ToImmutable();
```

- Extension methods can be used to create immutable collections from an `IEnumerable`:

```
var list = new[] { 1, 2, 3, 4 }.ToImmutableList();
```

Mutable operations of immutable collections are similar to the ones in regular collections, however they all return a new instance of the collection, representing the result of applying the operation to the original instance.

This new instance has to be used thereafter if you don't want to lose the changes:

```
var modifiedList = list.Add(5);
```

After executing the above statement, the value of the `list` will still be `{ 1, 2, 3, 4 }`. The resulting `modifiedList` will have the value of `{ 1, 2, 3, 4, 5 }`.

No matter how unusual the immutable collections may seem to a non-functional programmer, they are a very important building block in writing functional code for .NET framework. Creating your own immutable collection types would be a significant effort.

LINQ – Language Integrated Query

A much better known functional API in .NET framework is LINQ.

Although it has never been advertised as being functional, it manifests many previously introduced functional properties.

If we take a closer look at LINQ extension methods, it quickly becomes obvious that almost all of them are **declarative** in nature: they allow us to specify *what* we want to achieve, not *how*.

```
var result = persons
  .Where(p => p.FirstName == "John")
  .Select(p => p.LastName)
  .OrderBy(s => s.ToLower())
  .ToList();
```

The above query returns an ordered list of last names of people named John. Instead of providing a detailed sequence of operations to perform, we only described the desired result. The available extension methods are also easy to compose using the chaining syntax.

Although LINQ functions are not acting on immutable types, they are still **pure functions**, unless abused by passing mutating functions as arguments.

They are implemented to act on `IEnumerable` collections which is a read-only interface. They don't modify the items in the collection.

Their result only depends on the input arguments and they don't create any global side effects, as long as the functions passed as arguments are also pure. In the example we just saw, neither the `persons` collection, nor any of the items in it will be modified.

Many LINQ functions are **higher-order functions**: they accept other functions as arguments. In the sample code above, lambda expressions are passed in as function arguments, but they could easily be defined elsewhere and passed in, instead of created inline:

```
public bool FirstNameIsJohn(Person p)
{
  return p.FirstName == "John";
}

public string PersonLastName(Person p)
{
  return p.LastName;
}

public string StringToLower(string s)
{
  return s.ToLower();
}

var result = persons
  .Where(FirstNameIsJohn)
  .Select(PersonLastName)
  .OrderBy(StringToLower)
  .ToList();
```

When function arguments are as simple as in our case, the code will usually be easier to comprehend with

inline lambda expressions instead of separate functions. However, as the implemented logic becomes more complex and reusable, having them defined as standalone functions, starts to make more sense.

Conclusion:

Functional programming paradigm certainly has some advantages, which has contributed to its increased popularity recently.

With no shared state, parallelizing and multithreading has become much easier, because we don't have to deal with synchronization issues and race conditions. Pure functions and immutability can make code easier to comprehend.

Since functions only depend on their explicitly listed arguments, we can more easily recognize when one function requires a result of another function and when the two functions are independent and can therefore run in parallel. Individual pure functions are also easier to unit test, as all the test cases can be covered by passing different input arguments and validating return values. There are no other external dependencies to mock and inspect.

If all of these make you want to try out functional programming for yourself, doing it first in C# might be an easier option than learning a new language at the same time. You can start out slow by utilizing existing functional APIs more and continue by writing your code in a more declarative fashion.

If you see enough benefits to it, you can learn F# and go all in later, when you already become more familiar with the concepts ■

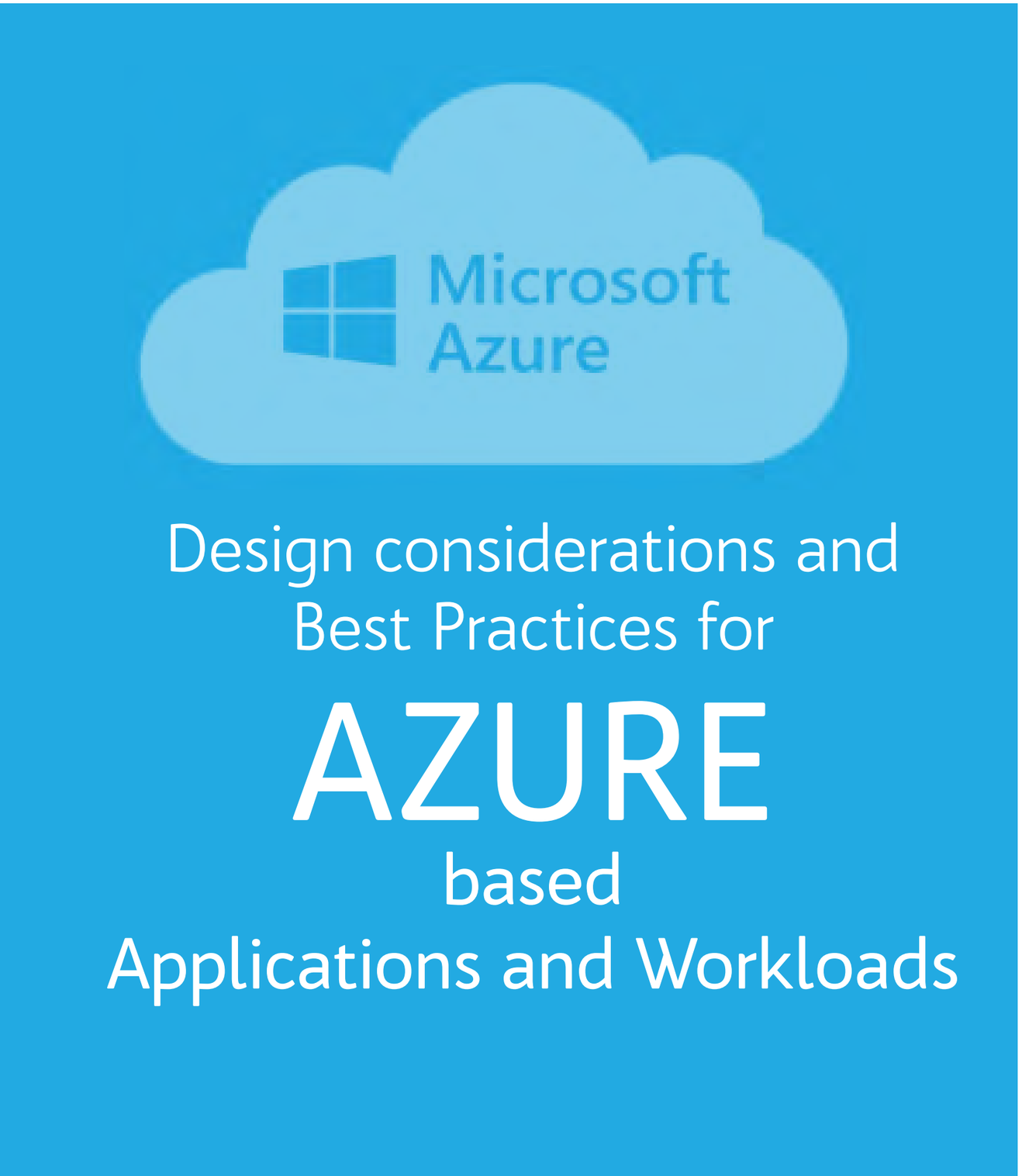
Microsoft® MVP Most Valuable Professional

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.

Vikram Pendse



Design considerations and Best Practices for AZURE based Applications and Workloads

There is a massive adoption of the Microsoft Azure platform across the world. Some enterprises/customers have been using it since its inception and have grown their products, services based on the platform over a period of time. There is also a large number of audiences for whom Azure is new and some of them have migrated to Azure from a different cloud platform like AWS (Amazon), Digital Ocean, IBM Blue Mix etc.

This article gives an overview of the end-to-end services that allow small customers as well as large enterprise to move their workloads on Azure smoothly. It also provides best practices to keep costs in control and ensure highly available scalable deployments.

Case Study

A mid-size Foo Solutions Ltd. has recently moved into Cloud Business and Solutions.

Their team has started offering managed services and consulting to their customers and are offering help to move on to Azure from any other cloud service.

A challenge in front of them is, their customers are demanding high availability solutions on Azure. Unfortunately, Foo Solutions Ltd. is relatively new in the cloud business, they rather have more expertise on traditional hosting platforms. So they are missing out on some basic Azure best practices, availability, security and monitoring guidance.

This article aims at providing Foo Solutions some best practices so that they can address their customers' needs and add more value to their managed services.

Understanding the design considerations and pain areas while building Azure based application

Here are some commonly acknowledged design considerations while building Azure based applications.

- Availability (Platform and Infrastructure)
- Dev and Test Environment
- Monitoring – Security and other advisory
- Cost

Now let us also state the very common workloads we usually see on Azure (more on the Microsoft stack but some of them can be part of Linux/open source ecosystem as well)

1. Virtual Machines
2. SQL DB (PaaS and IaaS)
3. Monitoring
4. Backup

Note: *Although "Storage" is a common component, since we are discussing more from a cost sensitive and rapid/ robust deployment point of view, hence will not consider storage here.*

Before we start discussing the above points, let us quickly understand the nature and types of Azure Subscriptions.

Understanding the types of Azure Subscriptions

There are various types of Azure Subscriptions available today. Each one has its own benefits and limitations.

Let us go through the commonly used Azure Subscriptions:

1. **Enterprise Agreement (EA)** – This is normally purchased by large enterprise customers where they give commitment for consumption of certain amount. Microsoft then adds a couple of value-add benefits on top of it, depending on type of customer, nature of business, country, region and policies.

2. **Pay as you go** – This is very commonly used by small startups, individuals as well as large enterprises. This is purely based on consumption and there is no pre-commitment for consumption. The amount is chargeable solely on the basis of actual consumption.

3. **MSDN** – This option is available if you are a MSDN subscriber and it comes with limited credit. Usually it is not used for deploying production workloads.

4. **Cloud Solution Partner (CSP)** – The Cloud Solution Partner is a newly launched program where one can get an Azure subscription from an authorized CSP qualified by Microsoft in that particular region. Pricing, Terms and conditions are different than that of Enterprise Agreement (EA).

Availability

One very common definition of "High Availability" is "Always available i.e. 24x7".

When we say so, there are a couple of perceptions and misunderstandings that are associated with this term. Most of the time, people fail to understand the *responsibility* aspect when they build *High availability* solutions. It is generally assumed that Microsoft is solely responsible for *High availability* which is obviously wrong to assume.

We will now see how we can achieve "High Availability" in Microsoft Azure.

While going for High Availability / Highly Available solutions, you first need to understand which bucket does your solution fall in - Infrastructure-as-a-Service (IaaS) or Platform-as-a-Service (PaaS).

Sometimes it might be a mixed solution as well.

While building high availability solutions, you need to ask two questions before designing your application:

1. What is the amount of downtime acceptable for a particular deployment?

2. If deployed resources/environment is down, then what happens next and what is the roadmap or action plan post this crisis?

High Availability in Azure IaaS (Infrastructure-as-a-Service) using "Availability Sets"

Virtual Machines (VM) are the core of Azure IaaS Services along with allied offerings of Virtual Networks (VNETs) and related connectivity patterns like P2S, S2S and Express Route.

There is a common perception that we always need high power CPU and large memory machines for better performance and availability. It does makes response time faster and reduces the need of any additional machine or load balancing.

However, in Microsoft Azure or for any cloud platform for that matter, there is always a possibility that VM can go down or is not available due to planned/unplanned maintenance.

Here are some very common issues due to which VM might not be available and an application on VM can have downtime or in some cases it is available, but not accessible or reachable.

- Disk failure – Very commonly observed problem for a VM.

- Host Machine Updates – Microsoft keeps on updating and patching their host machines. There is no defined schedule available publicly for this. But during update cycle, few VMs may face issues in booting or may boot up, but becomes inaccessible.

- Node failure – Node failure at the datacenter level.

- Power/Network issues at Datacenter – Power or Network failure at Datacenter side.

- Datacenter down – Very rare phenomenon but it can be down due to natural disasters.

Availability Sets allows you to put 2 or more virtual machines (usually with similar configuration). There can be maximum 100 VMs in an individual availability set. When you put your Virtual Machines (VMs) in availability set, you are bound to get availability around 99.95% as per the Microsoft's standard SLAs. Each of the machine is associated with Fault and Update Domain.

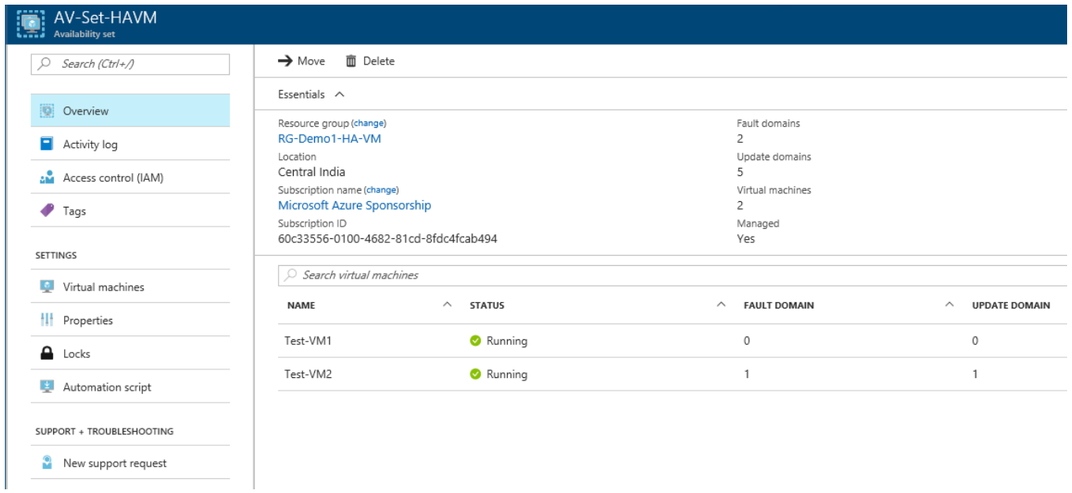
Let's understand what Fault Domain is and what Update Domain is.

Fault Domain – Failure due to Network/Power/Hardware Issues causing your VMs down/inaccessible.

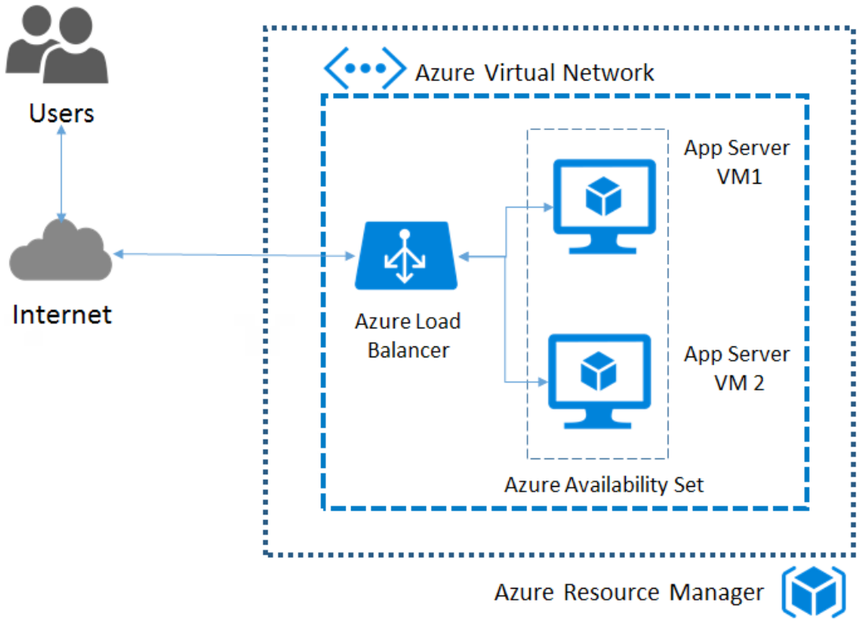
Update Domain – Guest VMs undergoes a planned/unplanned maintenance to update the patch of Guest VMs/Service Fabric bringing down your VMs.

Availability Sets addresses these issues with the help of Load Balancer. Load Balancers distribute the traffic and enables to keep minimum one instance active during plan/unplanned maintenance activities or failures.

This is how an availability set looks like:



Here is how its architecture looks like.



How you can verify whether Availability Set Configuration or HA is really working or not?

It is very simple to test. Once you have installed your application on both VMs, you can shutdown either of the VMs and observe how the request goes to second VM without having any downtime. You can customize the default landing page of your app and put some text in the header like "Running on VM1" or "Running on VM2" just to check if HA is working or not. You can remove the text once test is successful.

Is "High Availability" with Single Virtual Machine possible?

Earlier there were no SLA that single instance will not fail. Now since November 2016, Microsoft guarantees that it will be available for 99.9% with premium storage. This is very unique since no other Cloud providers provide such Single Instance SLA.

However single instance SLA and High Availability are two different concepts. Single instance SLA does not mean your solution is highly available. For High Availability, you still need Availability sets. If you need High Availability, then still go for Availability sets.

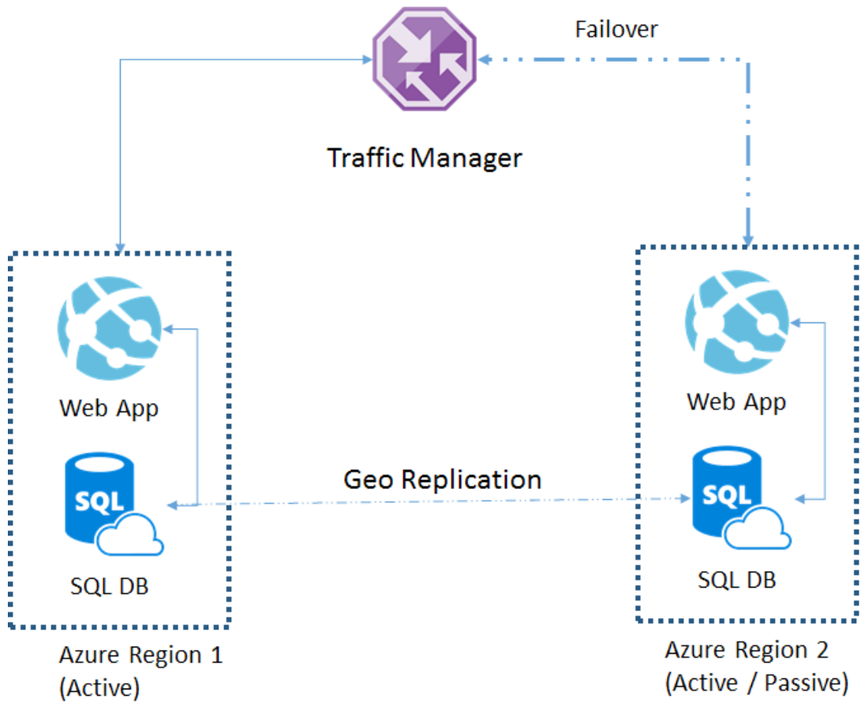
High Availability in Azure PaaS (Platform-as-a-Service)

In the previous section, we talked about the availability of an application hosted on an Azure Infrastructure. But what about applications hosted as Azure PaaS?

In Azure PaaS, the environment is multitenant and High Availability is built-in. Hence no additional configurations or efforts are required similar to what we have in IaaS to configure Availability Sets. Also, High Availability can be achieved using a single instance as Microsoft is responsible to maintain the SLA as a service provider.

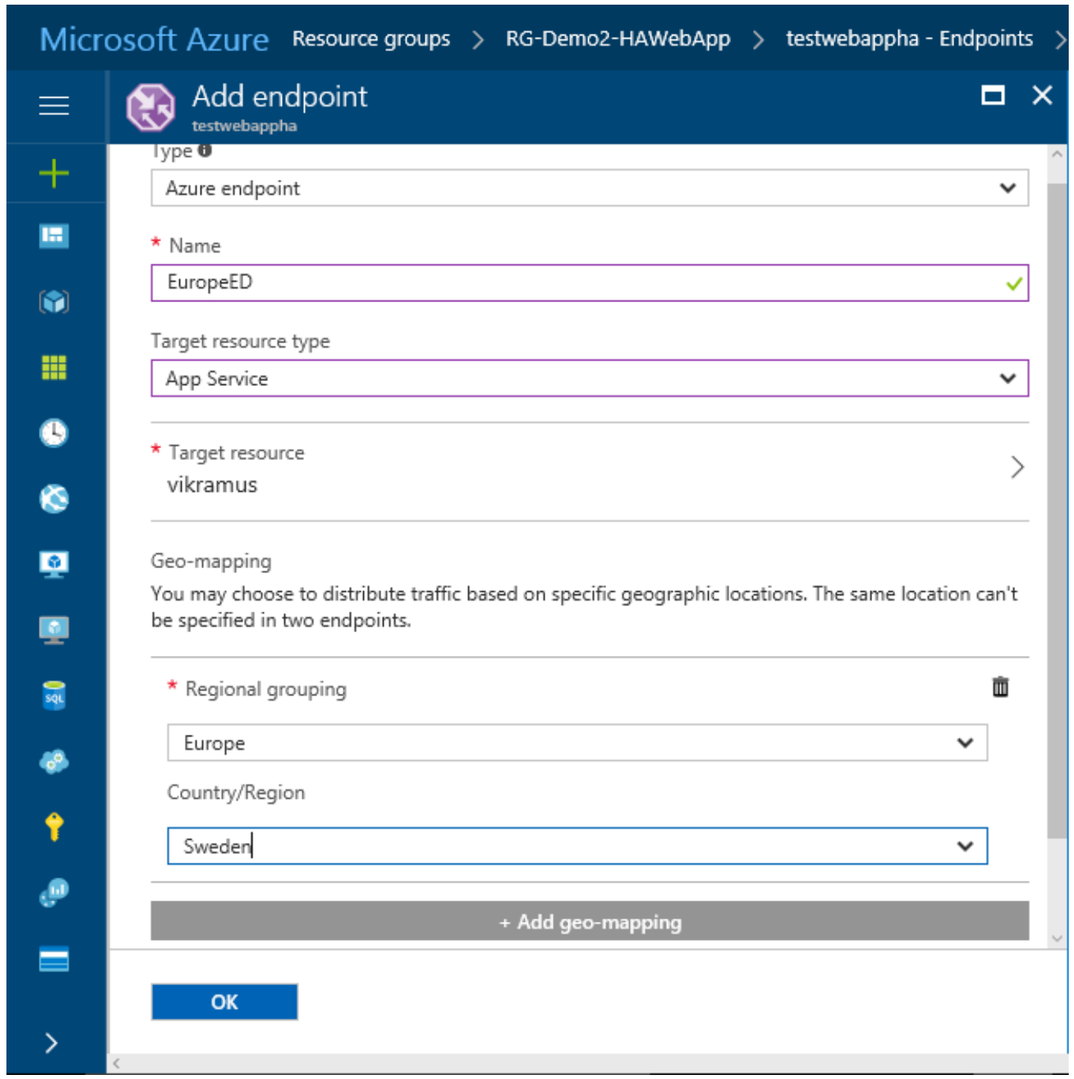
One of the common question which majority of customers asks "What happens when entire region goes down?"

Here doing Multi-site deployment can save you during an outage at a particular Data Center.



If you see the above architecture, we can see that we have deployed multiple instances of the application which are scattered across datacenters. Usually we tend to provision it in the same region to overcome latency issues, but if the problem is region wide, then it is good to have it in a nearby region, instead of having it in the same region.

Geo replication of data helps you achieve consistency. There are various data replication strategies available in Azure and you need to choose one based on your needs. The new Traffic manager policy allows you to configure distribution of your workload based on the geography as shown below.



In case of a failover, Traffic Manager will route the traffic to the most immediate available deployment in order to maintain availability and overcome the downtime of the application.

There are also couple of patterns which Microsoft has recommended like "Active-Active"," Active-Passive" etc. which are based on your data read/write and synchronization strategy. Usually these patterns come up with additional monitoring apps which does the job of probing and ensures which deployments are healthy and which are not.

Thus, you can maintain high availability in the platform-as-a-service kind of deployment strategy.

SQL PaaS and SQL IaaS – Which one to choose?

There are a majority of enterprises having one or the other flavor of SQL Server. It can be Professional or can be Enterprise or even can be Express in some cases.

Now-a-days due to awareness about cloud and data security, enterprises are keen to leverage benefits of the cloud platform and wish to save their infrastructure and maintenance cost.

There are two major SQL offerings in Azure which are SQL PaaS (also widely known by "SQL Azure/Azure SQL") and SQL IaaS.

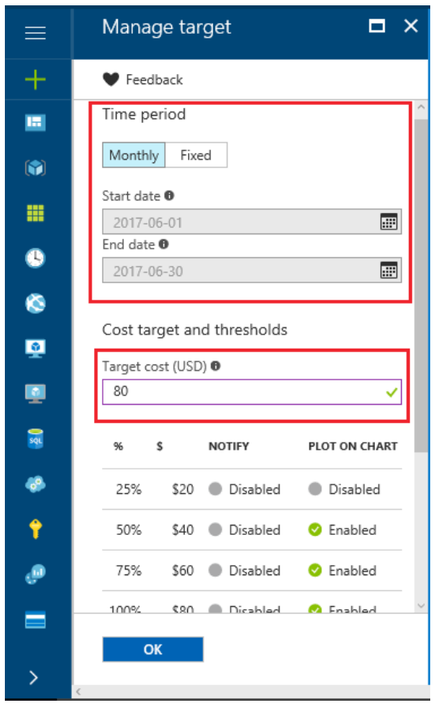
Here are some differences which will help you to understand which option is right for you and what you need to know while making a choice between SQL PaaS and SQL IaaS.

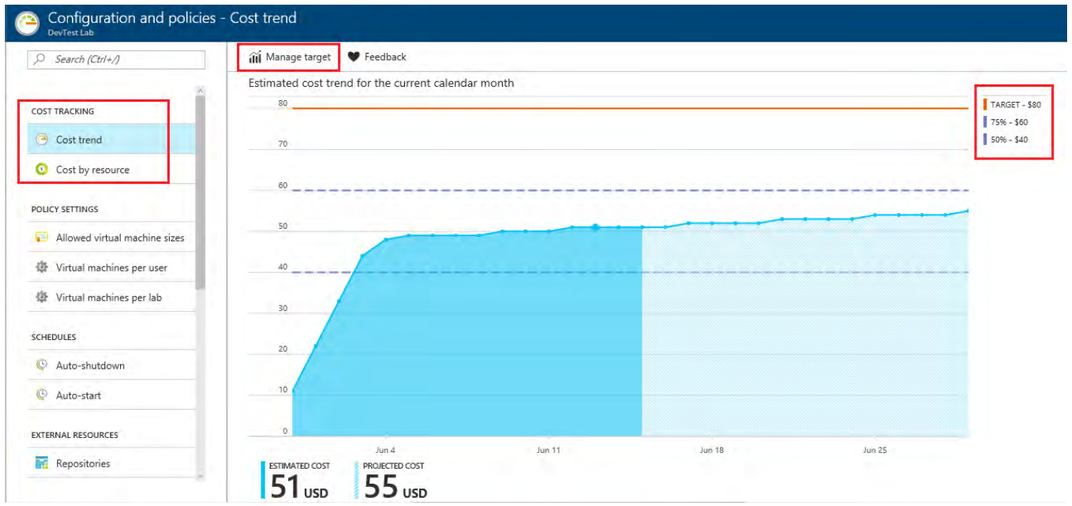
#	SQL Server on VM	SQL Azure Database
1	This is IaaS offering on Azure	This is PaaS offering on Azure. It is also termed as "Database as a service (DBaaS)"
2	Access to underlying VM is available	Access to underlying VM is not available and everything to be accessed over TDS (Tabular Data stream) based endpoint
3	Automated backups, DR and high availability is not available and one needs to configure it	DR, Backup and High availability is available by default.
4	Eliminates Hardware cost	Eliminates hardware and administration cost as well
5	Distributed transaction or all SQL server capabilities are supported	- Distributed transaction is not supported. - Additionally, there are restrictions on the usage of some reserved keywords also. - USE command not supported.
6	DB mirroring, Log shipping, transaction replication supported.	DB mirroring, Log shipping, transaction replication not supported.
7	SSIS, SSRS, SQL agent is available.	SSIS, SSRS, SQL agent is not available.

Dev and Test Environments

Dev/Test Labs is one of Azure's offerings which enables you to quickly spin up the Dev and Test environment in Azure. It helps to spin up VMs and also allows to store the template as "formulae" and provides reusability of a machine image.

It also helps to install the commonly required software/ tools as "Assets" via package management methodologies like "Chocolatey". Dev/Test labs allows you to implement RBAC (Role Based Access Control) on your environments and enables you to put the cap/restriction on the new IaaS component provision and puts control on overall deployment as shown below:





These are a unique set of features will allow enterprises to quickly provision the environment and can be integrated well with DevOps and CI/CD pipeline as well. So instead of provisioning single VMs or set of VMs one by one, this is a good quick and scalable option available for rapid infrastructure deployment with tons of good features. We have already covered this service in one of our articles here which you can refer to right away:

http://www.dotnetcurry.com/windows-azure/1296/devtest-labs-windows-azure

Monitoring – Security and other advisory

Monitoring is one of the core services which most of the time is overlooked by a majority of customers.

The primary reasons are cost, configuration and perceptions about these services. There are plenty of monitoring services available. There are rich logging solutions like Application Insights which gives you minute details about the behavior of your application.

For infrastructure monitoring, Log Analytics (OMS) is one of the very popular service in Azure, followed by a newly available service such as *Azure Monitor*.

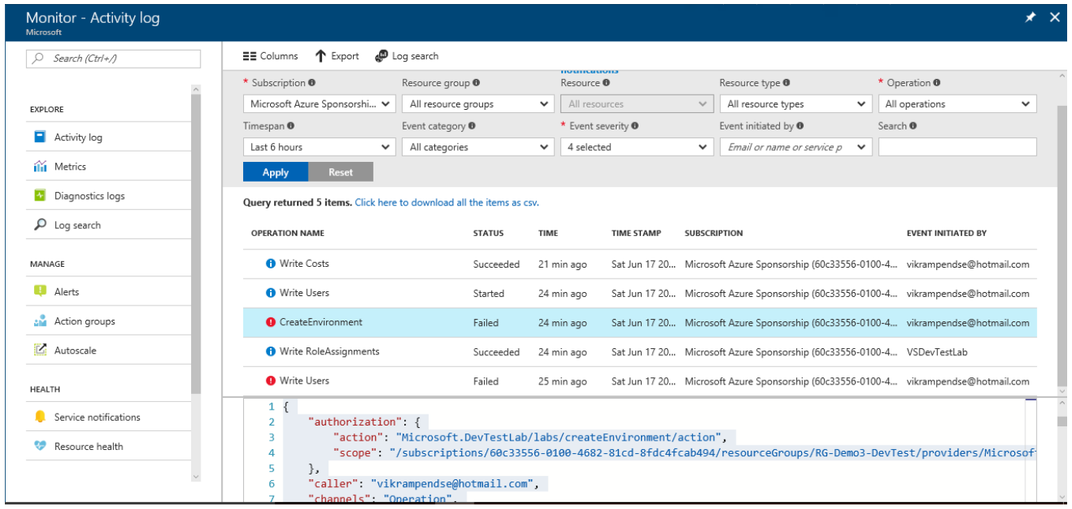
OMS give you deeper monitoring and add-ons to monitor your Active Directory (AD), IIS logs, Networks and even Containers too. It is very easy to configure and use as well.

OMS also helps you with various filters to search logs effectively and also allows to build data source for Power BI (Microsoft's interactive data visualization BI tool). Alert configuration and triggering mechanism is baked in, in these services.

We have already covered this service in one of our articles:

http://www.dotnetcurry.com/windows-azure/1310/log-analytics-oms-azure

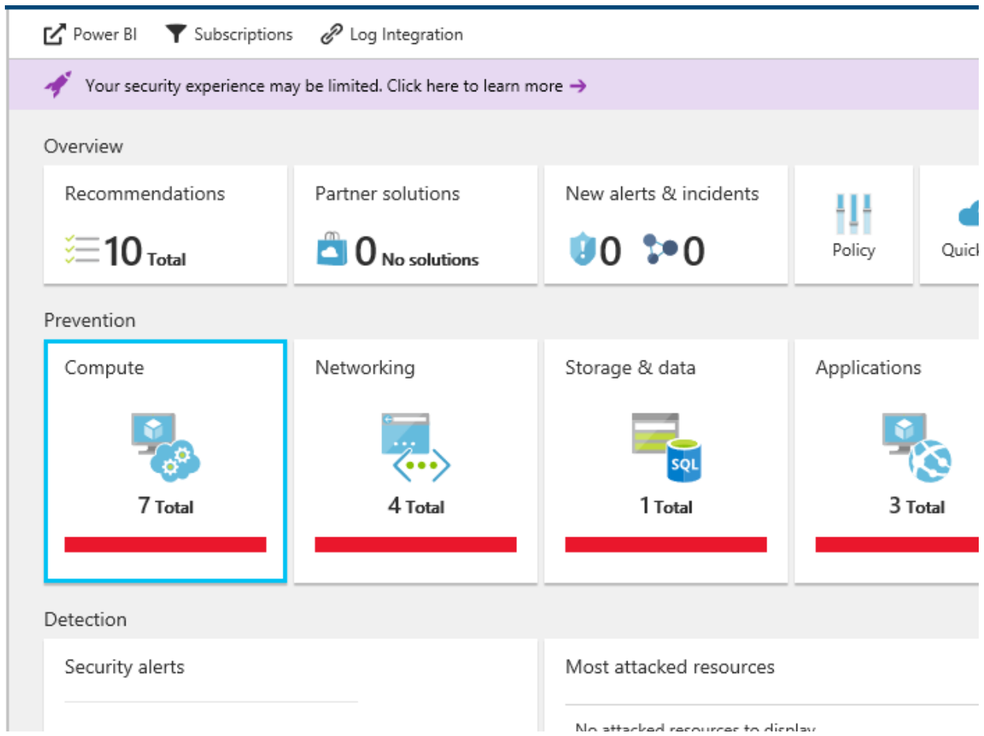
Here is a glimpse of Azure Monitoring services:

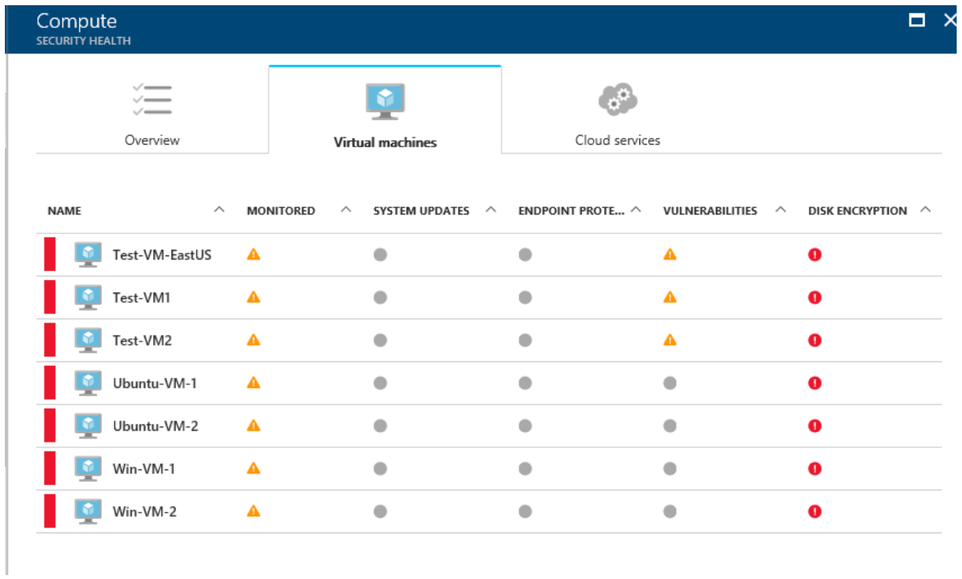


Although there are several ways to protect your infrastructure and application, Azure offers another service called "Azure Security Center" which helps customers to understand what is missing and what are the best practices to be applied on their existing infrastructure in order to make it more secure and stable.

Besides monitoring, we also need to know certain best practices for Security, Patches and overall updates. Moreover, there are value-add services and recommendation available to give you an illustrious ROI based on your consumption.

Here is a quick example of how the Security Centre looks like:

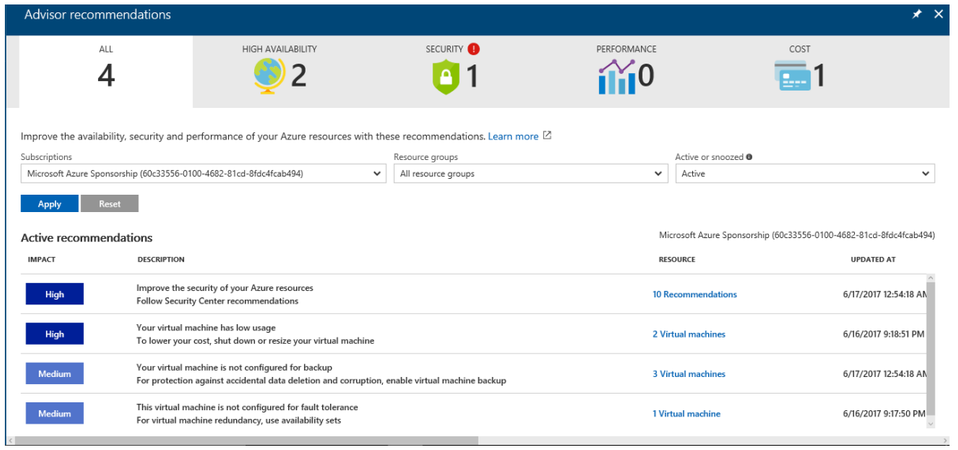




On the other hand, there is another service which acts as an advisoryfor your deployments in your subscription which is known as "Azure Advisor".

This service not only gives you best practices, but there are categories like "High Availability (HA)", "Security", "Performance" and "Cost" too. You can always look at the advisory and implement the suggestions accordingly. It always gives you the best suitable advisory based on your deployments and the way you have provisioned it.

Here is the dashboard for Azure Advisor service:



This comes in handy and you can keep monitoring this service daily.

Azure Monitor, OMS, Security Center and Advisory are all good to have services for your deployed resources, irrespective of the size, nature and count of resources you have in Azure.

Saving Costs

At the beginning, we saw various ways to get an Azure Subscription. Azure primarily works on "Pay as you go" model and hence "Cost" is one of the most critical aspect of any Cloud/Azure deliverable, since it is always directly proportional to your consumption cost.

We have already stated few best practices and service in the previous sections which can help you to identify bottlenecks on time, provide ease of configuration and deployment.

However, here is a summary as well as some additional best practices to save cost:

1. Use ARM template instead of Classic for all new Azure deployments
2. Azure DevTest labs can be used for quick infrastructure creation
3. Cost can be saved by using Capping and Restriction
4. Use Automation (In PaaS as well as in IaaS) wherever applicable
5. Use Scaling/VM Scale Set for auto scaling for IaaS deployments
6. Automation in start and shut down of VMs if they are idle for certain amount of time

Some additional value-add best practices in terms of monitoring and security:

1. Use Azure Advisory and Monitor for quick resolutions, recommendations etc.
2. Use OMS/Monitor for monitoring – for deeper monitoring
3. Use Azure Security Center for additional security guidance
4. Applying NSG on the network, VMs etc. – UDR, Inbound/Outbound rules
5. Identify the correct size and type of VMs

6. Futuristic design of VNET and Subnets – Ensure enough IPs are available in case additional VMs need to be provisioned in the specific subnet. One can always create a new subnet but good design can allow to accommodate new deployments rather than creating new one for each.

The above list covers most common scenarios and allows you to save cost by following best practices and getting into deeper interaction with Azure Advisory and Security Center.

Conclusion:

There is a significant growth in the Azure platform from the past few years because of the value add services in PaaS and IaaS. The end-to-end services features allow small customers to large enterprise to move their workload on Azure smoothly. Since there is always a cost associated with any Azure component, it is important to keep it in control and ensure the deployments are highly available, robust, secure and scalable in order to serve/cater end customers, flawlessly ■

Microsoft® MVP Most Valuable Professional

Vikram Pendse

Author

Vikram Pendse is currently working as a Technology Manager for Microsoft Technologies and Cloud in e-Zest Solutions Ltd. in (Pune) India. He is responsible for Building strategy for moving Amazon AWS workloads to Azure, Providing Estimates, Architecture, Supporting RFPs and Deals. He is Microsoft MVP since year 2008 and currently a Microsoft Azure and Windows Platform Development MVP. He also provides quick start trainings on Azure to startups and colleges on weekends. He is a very active member in various Microsoft Communities and participates as a 'Speaker' in many events. You can follow him on Twitter at: @VikramPendse

Thanks to Kunal Chandratre for reviewing this article.

Gerald Versluis

Using MVVM in your Xamarin.Forms app

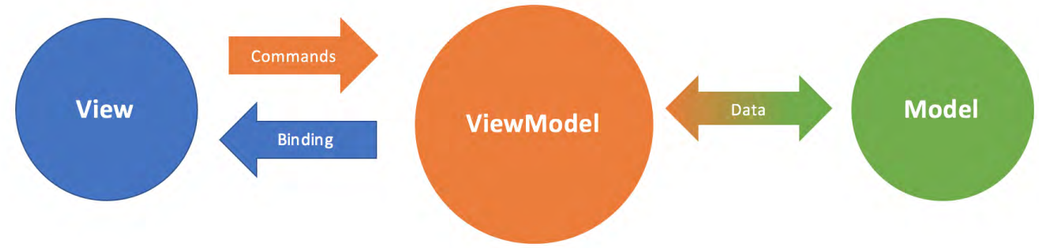
If you have looked into Xamarin.Forms, or even built an app with it, you might have noticed that Forms has everything to implement the MVVM framework. Xamarin. Forms is already equipped with data-binding, commands and dependency injection which are some MVVM ingredients we will look at in this article.

What is MVVM?

Let me briefly explain what MVVM is, in case you don't know it yet.

MVVM stands for Model-View-ViewModel. It is derived from the MVC pattern which you probably already know about. It is an architectural pattern which enables you to separate business logic from UI code. The pattern consists of three ingredients: Model, View and ViewModel.

The ViewModel acts as a value converter between the Model and the View. The image below shows a schematic representation.



The ViewModel is responsible for converting values in a way, for the View to present them to the user. To help you update the UI from the ViewModel, data binding is used.

MVVM with Xamarin.Forms

Now, let us zoom in on how MVVM can be used in Xamarin.Forms apps.

If you have built a Forms app before, you will know that one of the choices you have to make early on is to use either XAML (Extensible Application Markup Language) OR code to assemble your UI. Both have the same functionality and can do the exact same thing, it just comes down to a matter of taste.

I like to use XAML as I think it is clearer. This is why for the rest of this article, I will use XAML. However, this choice does not mean you can only use the option you have chosen. If you decide later on to switch to code, you can and vice versa too. If you would like, you could even mix both, although I would not recommend it.

XAML is created by Microsoft and is used extensively in Windows Presentation Foundation (WPF) applications. Although the concept is identical, the naming between these two platforms is different. This probably is due to two main reasons: first, the naming of Xamarin XAML is adapted to be better suited for mobile concepts, and second; it prevents people from taking existing layouts in WPF and paste them into a mobile app, causing the app to be a user experience disaster.

But Xamarin.Forms has more in store for you when you want to work with MVVM. It also has data binding and commanding. We will learn about this later on.

One last thing on naming; throughout this article, I will refer to a ViewModel as a PageModel. There is no difference, it is just that a View in Xamarin (or even mobile) is refered to as a *Page* most of the time. To make it more clear – and simply because I have become more accustomed to it – I always use Page and PageModel.

Creating a Sample App

The best way to learn about something is to get to work with it, so let's create a sample app which shows us how to implement MVVM.

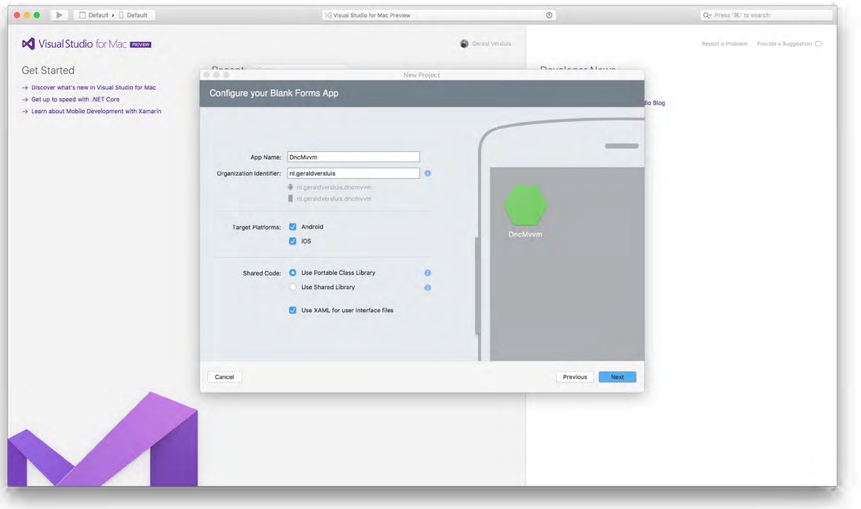
In this example, I will be working with Visual Studio for Mac, but everything will work just fine on Windows with Visual Studio as well, and for building Universal Windows apps (UWP).

Let's first create a new Xamarin.Forms app. Go to *File* > *New Solution* and choose the *Blank Forms App*. Make

sure you don't choose the *Forms App*, which is a great template as well as it gives you an ASP.NET Core backend automatically, but that is not what we need right now.

In the next screen, name your app and make sure to choose a *Portable Class Library* for 'Shared Code' (I will refer to this as PCL from now on) and check *Use XAML for user interface files*. The last option, use XAML, is just to setup a first page for you. If you forget to check it, you can still *use XAML* in your project, just like you can still use code when you *do* use XAML.

The last screen presents you with some options to configure a repository and add a test project. We will not focus on this right now, so you can leave these options as-is.



After Visual Studio has finished processing, you should be presented with your brand-new app. I have named mine DncMvvm. Because we choose to use XAML, there is a page already in our PCL which is named after our project, so in my case *DncMvvmPage.xaml*.

When we open this up, there is a very simple ContentPage in it with a centered Label. We can see how it looks like in the Xamarin.Forms XAML previewer. This is a very powerful tool when designing layouts, be sure to check it out.

In this page, we will insert our first bit of MVVM and data binding.

First, let's create a new class which we will be named *DncMvvmPageModel*. While naming is not important, for overview, it can be helpful to stick to a pattern of *NamePage* and the accompanying *NamePageModel*. This way you can easily see what PageModel corresponds to which Page. This will have another advantage when we are implementing a MVVM framework. This is something I will show you later in this article.

The implementation of the DncMvvmPageModel is simple; it holds one public property of type string. That is it, for now.

```
public class DncMvvmPageModel
{
  public string LabelText { get; set; }
}
```

To implement this, now go to the code-behind of our page, create a new instance of the PageModel and use it as such.

```
public DncMvvmPage()
{
  InitializeComponent();

  var pageModel = new DncMvvmPageModel();
  pageModel.LabelText = "Hello from the PageModel!";

  BindingContext = pageModel;

}
```

As you can see, a page has a property *BindingContext*. By assigning any type of object to this property, it will function as the PageModel for that page. You can then bind to any public property that is in the PageModel (DncMvvmPageModel in this case).

In the above code, I created an instance of the PageModel, assigned a text to the LabelText property and set the PageModel as the BindingContext for our page.

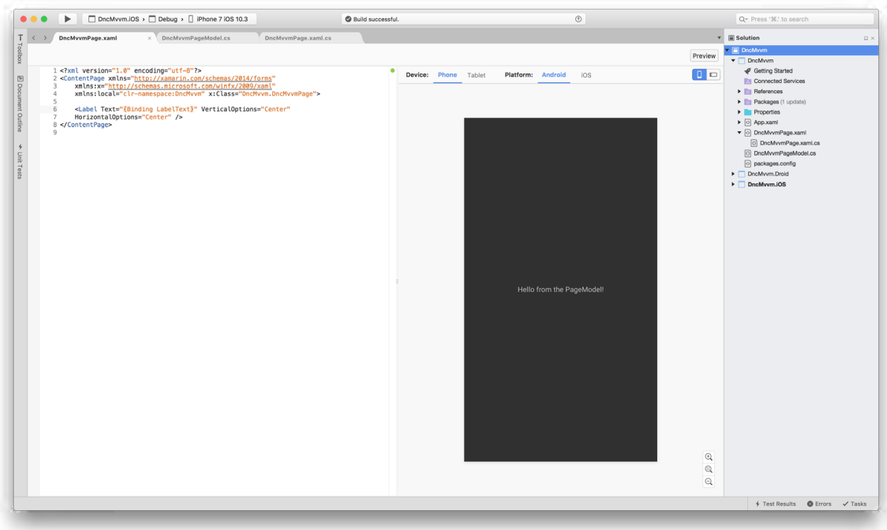
Note: This is not a best practice. The code-behind should be clear of any code as much as possible. With this sample, I just want to show you that you can set the values in your PageModel from anywhere.

There is now just one last thing to do, set the Label in the XAML of our page to use the property of our PageModel.

We replace the static text that was defined earlier like this:

```
<Label Text="{Binding LabelText}" VerticalOptions="Center"
HorizontalOptions="Center" />
```

When the app is run now, you will see that the text comes out of our PageModel. This is also reflected in the XAML previewer as you can see in the figure below.



I have divided the source code into release on my GitHub repository, you can find this first step here: https://github.com/jfversluis/DncMvvm/releases/tag/step-1.

Commands

Showing data from our PageModel is one thing, but as you can imagine, we would also like the ability to invoke some code.

Traditionally, we would do this by the means of events. However, using *events* requires a tight coupling between the Page and the code-behind, which is not what we want. Therefore, Commands were invented. *Commands* are basically another type we can bind to in our PageModel, but can execute code for you.

Let us have a look at how this works. First, go to the PageModel, and add a new property of the *Command* type. In the code below you see the updated PageModel.

```
public class DncMvvmPageModel {
  public string LabelText { get; set; }

  public Command KnockCommand { get; }

  public DncMvvmPageModel() {
    KnockCommand = new Command(() => {
      LabelText = “Who’s there?”;
    });
  }
}
```

There is now a new property which I called *KnockCommand*. In the constructor of my PageModel, I assign it with a handler. In the Command, I set the LabelText to a different value and expect it to be updated. To hook up our new property, go back to the XAML page and define a button which binds to our KnockCommand property. I have juggled around with the layout a little bit to make it look better as well.

```
<?xml version=”1.0” encoding=”utf-8”?>
<ContentPage xmlns=”http://xamarin.com/schemas/2014/forms”
  xmlns:x=”http://schemas.microsoft.com/winfx/2009/xaml”
  xmlns:local=”clr-namespace:DncMvvm” x:Class=”DncMvvm.DncMvvmPage”>

  <StackLayout VerticalOptions=”Center” HorizontalOptions=”Center”>
    <Label Text=”{Binding LabelText}” />
    <Button Text=”Knock, knock...” Command=”{Binding KnockCommand}” />
  </StackLayout>
</ContentPage>
```

When running this code now, we see the new button and when we press it, well nothing happens!

Although if we put a breakpoint in the code of the Command, we would see that the code is triggered. To update values in the View (or Page) from the PageModel, there is something else we need to implement. The code for implementing the Command is the second release I have created in the repository.

INotifyPropertyChanged

To update values in the UI from a PageModel, there is something we need to implement. This is the *INotifyPropertyChanged interface*. With the implementation of this interface, you can notify the UI (in our case defined by XAML) that a value has changed, and needs to be retrieved from the PageModel again.

Implementation is very simple, but requires some refactoring. Shown here is the code after refactoring the interface.

```
public class DncMvvmPageModel : INotifyPropertyChanged {
  private string _labelText;
  public string LabelText {
    get
    {
      return _labelText;
    }
    set
    {
      _labelText = value;
      PropertyChanged?.Invoke(this, new
      PropertyChangedEventArgs(nameof(LabelText)));
    }
  }
  public Command KnockCommand { get; }
  public DncMvvmPageModel() {
    KnockCommand = new Command(() =>
    {
      LabelText = “Who’s there?”;
    });
  }

  public event PropertyChangedEventHandler PropertyChanged;
}
```

As you can see, I have declared the interface after the class name. This interface forces me to implement a new public event of the type *PropertyChangedEventHandler*. Our binder will check to see if this interface is implemented and hook up this event automatically.

To raise this event, I needed to refactor the LabelText property to a property with a backing field. This way, I can invoke the event in the setter of this property. Now, when we run this code again, the text in our label *will* change!

You can imagine that as your project grows, the number of properties will grow and there will be a lot of repetitive code which just raises a flag whenever a property value is changed.

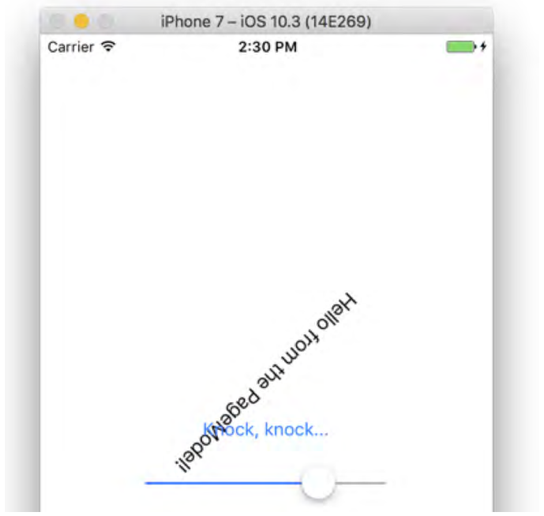
There are a number of ways to overcome this.

One is to introduce a generic base class which handles a lot of duplicate code for you. Another good option, which I tend to use almost always, is to look at the *Fody.PropertyChanged* (https://github.com/Fody/PropertyChanged) NuGet package. This injects the *INotifyPropertyChanged* code at compile-time for you. I will not handle the details in this article, but be sure to check it out.

This example only implements the INotifyPropertyChanged manually. The code for this can be found in the third release on the sample app repository.

A Bit Deeper into Data Binding

To dig a little bit deeper into data binding, we will look at how we can bind the property of one control, to the property of another. To demonstrate how this works, we will add a *Slider* control to our UI and let the value of that control adjust the Rotation of our Label control. The end result is shown in the image below.



Because we will now impact one control from another, we just need to adjust the UI code, and thus our XAML. You will see how to do it in the following code. For brevity, I have only included the StackLayout containing the controls, instead of the whole page.

```
<StackLayout VerticalOptions="Center"
HorizontalOptions="Center">
  <Label Text="{Binding LabelText}"
  Rotation="{Binding Source={x:Reference
  AngleSlider}, Path=Value}" />

  <Button Text="Knock, knock..."
  Command="{Binding KnockCommand}" />
  <Slider x:Name="AngleSlider"
  Maximum="180" Minimum="0" />
</StackLayout>
```

At the bottom, I have added a slider and I gave it a name. The minimum and maximum value are set to 0 and 180 which are degrees that the Label can be rotated at.

In the Label, I have declared the *Rotation* attribute. Instead of the binding syntax we have seen until now, there is something different here. I will dissect it for you.

A binding declaration like `{Binding PropertyName}` is a shorthand for writing `{Binding Path=PropertyName}`. The *Path* parameter specifies the property of the BindingContext to look for a value.

But there are more parameters you can use.

One of them is *Source*. With Source, you can specify a data source. With this parameter, you can specify a different source than the BindingContext which is specified for the whole page. In this case, we set the source to another control. This is why we needed to give the Slider control a name, so we can assign it as a source.

With this in place, we can run our app and when the slider is changing value, the label will rotate accordingly.

You can imagine how powerful this is. You can not only bind to PageModel values, but also to values of other controls or to values on the page. This all makes for a very flexible and extensible technique creating no coupling between your Page and any business logic. Remember; the PageModel is responsible for pushing data back and forth.

The code for this phase of the app can be found in the fourth release on GitHub.

Value Converters

One last thing I want to show you is *Value Converters*. In the UI, a lot of properties will just show strings, which is easy. But later on, you might want to give a control a certain color or show an image depending on a certain value, etc.

I will explain this with a real-life example.

I was building an app which had to do with gaming. The app showed games with their respective platforms. Each platform had its own color associated to it. There are multiple ways to solve this – as always – but I chose to do it with a value converter. With a converter, you can transform a data bound property from one type to another. In this case, I took in a string, which specified the platform like PS4 or Xbox and transformed it to a color.

Let me show you this in our sample app.

I have updated my XAML page and bound my Label's TextColor attribute to the LabelText property of my PageModel. I have done so because I want the color of the text to change based on this string value.

```
<Label Text="{Binding LabelText}"  TextColor="{Binding LabelText}"
Rotation="{Binding Source={x:Reference AngleSlider}, Path=Value}" />
```

Of course, we can't assign a string value to an attribute that expects a Color. This is where the ValueConverter comes in.

To create a converter, we need to create a new class which implements the IValueConverter interface.

```
public class WhosThereToColorConverter : IValueConverter
{
  public object Convert(object value, Type targetType, object parameter,
  CultureInfo culture)
  {
    var stringValue = value as string;
    if (string.IsNullOrWhiteSpace(stringValue))
    return Color.Black;

    if (stringValue.ToLowerInvariant().Contains("who"))
    return Color.Red;

    return Color.Black;
  }

  public object ConvertBack(object value, Type targetType, object parameter,
  CultureInfo culture)
  {
    throw new NotImplementedException();
  }
}
```

By implementing this interface, you need to define a *Convert* and *ConvertBack* method. These methods are responsible for supplying a new value in the right type. In this case, I get the string value, which we need to cast because a conversion can be from any type, and based on what the string contains, I determine a Color.

In the above code, I check to see if the text contains 'who' and then change the color to red.

In many cases, ConvertBack need not be implemented. When you only need to display a value in a specific way from your PageModel to your Page, you only have to implement Convert.

When you create a TwoWay binding (the value is updated in the PageModel when you edit it in the Page), you will need to also implement the ConvertBack method to transform the value from the one shown in

your UI, to a value that the PageModel understands. So, it is effectively the inverse of the Convert method.

The last thing we need to do is to tell the binding to use the converter.

Update the XAML for the Label to this:

```
<Label Text=”{Binding LabelText}”  TextColor=”{Binding LabelText,
Converter={StaticResource WhosThereToColorConverter}}” Rotation=”{Binding
Source={x:Reference AngleSlider}, Path=Value}” />
```

Notice how I added a converter parameter to the binding. This converter points to a static resource. There are multiple ways to define a static resource, but the simplest way is to define it in the same page. You can do so like this:

```
<ContentPage.Resources>
  <ResourceDictionary>
    <local:WhosThereToColorConverter x:Key=”WhosThereToColorConverter” />
      </ResourceDictionary>
</ContentPage.Resources>
```

The Key has to correspond with the *StaticResource* parameter on the binding. When we run this code and press the button, the text will change to ‘Who’s there?’ and because that contains ‘who’, the color will also change accordingly.

The code for this can be found in the fifth release on the repository.

Using a MVVM Framework

A lot of people have already gone and implemented the MVVM pattern before you. As such, there are also a lot of frameworks out there that can help you with this. Not only will they make your life easier, they also provide you with even more power and decoupling of your code.

There are multiple good frameworks out there, even from before the time that Xamarin.Forms was invented. They include, but are not limited to:

- MVVM Light
- Prism
- Caliburn.Micro
- and the very popular MvvmCross.

All these frameworks do a great job, and basically provide you with the same thing, they just have their own accent on certain aspects.

The framework I like to use is FreshMvvm (github.com/rid00z/FreshMvvm) by Michael Ridland.

This framework is very easy to use and very lightweight. Because Xamarin.Forms already has so many aspects of MVVM already implemented, a few optimizations are all we need. That is exactly what was in mind when FreshMvvm was designed. This framework was built specifically for Xamarin.Forms and therefore the naming is done appropriately. Everything is named with Pages and PageModels, while the naming convention with View will also work.

FreshMvvm as a MVVM framework

In this last part of my article, I will show you what is involved in implementing FreshMvvm as a MVVM framework and how it helps you optimize your code even further.

Installing is very easy, just install the NuGet package in your PCL project and that is it.

Remember how I told you that it would be convenient to keep the naming convention consistent?

FreshMvvm works by this naming convention. Automatically a PageModel which could be named PersonDetailsPageModel will look for a page called PersonDetailsPage.

Also, notice how I say that the PageModel will look for a page, FreshMvvm supports PageModel-to-PageModel navigation. This means you state to which PageModel you want to navigate to, instead of referencing an actual page. This helps you to further separate your UI from other code, because now, when you want to replace a Page, just create a new one, give it the right name according to the naming convention and your new Page will be shown without touching any code at all.

While this works out of the box, you can configure this differently if you want.

For all of this to work, your PageModels now need to inherit from the *FreshBasePageModel*. Go into the DncMvvmPageModel and let it inherit from this base class. Now go into the App.xaml.cs file and replace the assignment of the *MainPage* property.

```
MainPage = FreshPageModelResolver.ResolvePageModel<DncMvvmPageModel>();
```

This way you can resolve the page that is associated to this PageModel and it is shown as the main page. When running this, it doesn’t go right because we are still setting the BindingContext from the Page’s code-behind, remove this code to make it work as it should. If done right, you do not need any code in your pages anymore, at all. Just remember that now you need resolve your pages by the PageModel to make this automatic binding work.

FreshMvvm has the most basic page types already built-in: TabbedPage, MasterDetailPage and just a ContentPage are among them. They all work with this automatic building and you can specify PageModels which will automatically be resolved. For instance, if you want to have a page with multiple tabs, you would specify it like this:

```
var tabbedPage = new FreshTabbedNavigationContainer();
tabbedPage.AddTab<DncMvvmPageModel>(“Home”, “icon.png”);
tabbedPage.AddTab<AnotherPageModel>(“Another”, “anothericon.png”);

MainPage = tabbedPage;
```

To navigate from one page to another, you can use the methods that are exposed through the FreshBasePageModel. In the *CoreMethods* there are a lot of handy functions you can leverage among them, for navigation purposes. You can navigate to another PageModel by executing:

```
CoreMethods.PushPageModel<AnotherPageModel>();
```

There are some overloads as well which let you supply any object with which you can send values back and forth.

For instance, imagine you are building an app which manages contacts. You can select a contact from the list which brings you to a detail page. You could supply the object with the contacts details to the PushPageModel method and fetch it in the ContactDetailPageModel. You would do so by overriding the Init method. This method will be invoked when a PageModel is navigated to. You can check there if an object is supplied and handle it in your target PageModel. There is also the *ReverseInit* which does the opposite, you can retrieve a value when a PageModel is popped.

I have added some sample code for this in the repository in the sixth step for you to examine. If you would like to know more, please do not hesitate to contact me.

Final Thoughts

In this article, I have shown you what MVVM is, some MVVM concepts and what you can gain from it. We have seen that Xamarin.Forms is already very well prepared for the usage of MVVM. With the concepts of data binding, INotifyPropertyChanged and value converters there is very little needed to create clean and separated code.

To even take it a step further and optimize your development process, you can look at a framework, which helps you to implement MVVM in your app and takes some tedious tasks out of your hands.

I hope you will take the learnings from this article and implement this in your apps. If you have any questions, please reach out to me on Twitter (@jfversluis). All code that has been used can be found on my GitHub page https://github.com/jfversluis/DncMvvm. If you look under Releases, you can follow along step-by-step

Download the entire source code from GitHub at
bit.ly/dncm31-mvvm

Microsoft MVP Most Valuable Professional

Gerald Versluis
Author

Gerald Versluis (@jfversluis) is a full-stack software developer and Microsoft MVP (Xamarin) from Holland. After years of experience working with Xamarin and .NET technologies, he has been involved ina number of different projects and has been building several apps. Not only does he like to code, but he is also passionate about spreading his knowledge - as well as gaining some in the bargain. Gerald involves himself in speaking, providing training sessions and writing blogs (https://blog.verslu.is) or articles in his free time. Twitter: @jfversluis Email: gerald@verslu.is Website: https://gerald.verslu.is

Thanks to Damir Arh for reviewing this article.

Mehfuz Hossain

Bot Platform and Its Various Components

The Chatbot market is growing rapidly, it is already a 1.4 trillion dollar industry and there are over 100K bots on Facebook alone. AI chatbots can be used to arrange your travel plans or can become your smart matchmaker.

However, there is a degree of development knowledge needed to create an intelligent, cross-channel and scalable bot.

A Bot lives on top of existing communication channels that take the advantage of the underlying native UI and adds intelligence to make the communication between the software and an individual human, alike.

Bot development differs from app development in many ways.

Apps require proper planning, a specific designer for UX flow and platform specific building blocks. Bots on the other hand offers a more streamlined and linear execution.

Moreover, apps have become exhausting. People want fewer apps now. ChatBots can provide just enough UI for a consumer to get what they want, and as fast as possible.

However, building a bot isn't as simple as it looks.

Let's take a look at how bot development is done today and a new emerging development method.

Building a Bot from Scratch

In order to build a bot from scratch, apart from preparing conversational script and basic flow on how it should react to user interactions; you need to consider the various channels (Facebook, Slack, Skype, Viber, SMS, etc.) which you want to target.

You are responsible for preparing your development environment, add channel specific implementations, webhooks and research on your right cloud infrastructure, build and deploy the package.

Specs & Conversational Script

The conversational script acts as a use-case of actual user conversation. Because most chatbots interact with users, the bot must guide the user towards the answer or goal towards the end of the conversation. The content of the script creates a context that relies on user behavior and uses Natural Language Processing (NLP) to better handle the flows.

System Environment & Configuration

Next you need to create the architecture and design of the bot.

You are responsible for creating both the backend for managing the conversation data and providing baseline infrastructure of the bot and integrations to target the bot for multiple channels, which you want to target.

A developer will typically need to:

- spin up their own servers,
- build, deploy and manually wire up the corresponding SDKs,
- incorporate NLU (Natural Language Understanding),
- fiddle with platform specific API,
- manage hosting and
- preparation of deployment package including processing of dependencies, test and deploy.

These steps usually takes developers a bit to set up since there are many connecting interfaces, services,

plugins and configurations.

Will you manually host it yourself, use Azure or Amazon AWS (which requires configurations) or another service like Heroku? Will you create a developer dashboard to monitor your analytics data and health status?

Bot Development

With your system configuration and development environment in place, it is time to build your bot.

Use your conversational script to create a functional bot while debugging it to make sure that it responds to user interactions correctly and drives to the expected goal. This step is very straightforward, as your spec guidelines gives you the skeleton of the bot.

Deployment

Once you are satisfied with your bot, it is time to deploy it to a hosted environment. You need a stable hosting environment where the bot will reside, and that allows you enough flexibility to expand your hosting and monitor your resource utilization while keeping the cost low.

Integration to Messaging Channels

To get people to use your bot, you need to distribute it.

Pushing your bot to various messaging platforms such as Facebook Messenger or Slack gives people a chance to discover and interact with it. Platform such as Facebook Messenger lets you discover your bot from their integrated store in message app.

One of the key disadvantage of **Build Your Own Bot (BYOB)** approach is that it is your responsibility to build and test the integration endpoints while keeping up with the growing platform and features, or any breaking changes. Either you write your own ***If-else*** logic to facilitate the integrations or spin up a different app for each channel, while you duplicate your core implementation.

It is your job as the developer to monitor and manage each environment, conversations, figure out integrations and storing conversational data and so forth. Not to forget versioning. As the number grows, it becomes more work than actual bot development.

Moreover, in order to improve the response of these bots, you have to take a look at how users interact with it. By examining potential scripts, you can add them into your bot via an update to the publishing channel.

The Entire Process Looks Somewhat Like This:

Bot Spec & Script > System Configuration and Hosting Environment > Bot Development (actual coding) > Testing > Deployment > Publishing > Monitor & Analyze > Update > Repeat for each Bot

Now while the step-by-step process looks simple, there are hidden struggles within.

Jake Bennett, CTO at Seattle-based agency POP, tried creating a serverless app using AWS Lambda. So even if you don't host your own application, it isn't as easy as it seems. He details his struggles in a VentureBeat article, What I learned Building Serverless Apps . His main challenges was the time spent prior to building the application. It was the time spent configuring, creating and setting up the Lambda functions.

Configuration takes significant amount of time

To develop an app, Jake had to configure the environment for uploading the code, configure the Lambda function, create the API endpoint, set up the IAM security role for the function, configure the behavior of the HTTP request, configure the behavior of the HTTP response, create the staging endpoint and deploying the API.

This time should be factored in when estimating the project completion time since Lambda functions are nano-services (same if you use other services such as Azure).

Requires Domain Specific Knowledge

The biggest challenge for Jake was figuring out how to solve problems popping up, as the lack of documentation, made for wasted time spent researching issues. "Error messages are often cryptic and the number of configuration options is large, making diagnosis time-consuming." In short, setting up infrastructure is hard and requires domain specific knowledge.

How to balance between tight cohesion and loose coupling in structuring the app

Before the adoption of serverless architecture, functions were thought to exist in a larger cohesive application with shared memory and a call stack.

However, there is a lack of architecture design patterns for serverless applications. "With AWS Lambda, you are literally deploying a single function, outside the context of a larger application.

So how do you live without the things we take for granted every day like shared functions and common configuration settings?

Will you impose every function as a Lambda and create tons of configuration thus affecting performance (since every function invocation will be out-of-process call) or deploy a single Lambda function that acts as a controller for the entire application (you have to oversee the controller function in order to keep it efficient and manageable)?

Depending on your application, you may need a mix of the two but time spent on architecture can be costly and long.

So if we go back to the process map, even with serverless hosting, you can see how it could take a developer some time to build the bot from end-to-end.

Bot Spec & Script > Lamdba and API gateway configuration, Hosting Architecture > Bot Development (coding the bot) > Testing > Deployment > Publishing > Monitor & Analyze > Update

Bot-as-a-Service to Improve Productivity

With the era of popularity of chatbots, bot platforms are starting to spring up to make development workflow streamlined and simple.

There are many kind of bot platform ranging from tools, infrastructure setup, development frameworks to deployment and publishing platforms to speed up the process.

What all these services and products offer is to take the pain from the setup, configuration, building and deployment steps, so anyone can publish a bot into a messaging platform **as quickly as possible**.

How Bot Platforms Saves Time

The innovator's dilemma is searching for new approaches to past problems.

How can we optimize the current approach in order to shorten the time and gap so distribution of knowledge and resources can better reach the intended audience?

The platform approach takes the optimization level and redefines it into solving the same problem by a new solution (it's a paradox in itself).

Bot platform enables creation of more bot inventory (in the market) without using and wasting more resources (time, manpower, computer processes). The logical thinking is that if we empower and permit developers to build faster and more efficiently; more quality bots will be available on the market and thus letting more people use bots (increasing developer interest and starting the cycle over).

Bot platforms handle the infrastructure and system architecture setup (from stack creation to creating nano-services) while providing both a framework and platform for test, debug and deploy. Overall, monitor bot usage and analytics in one single dashboard and provide integrations and version control in case of catastrophic events either caused by developer or regression.

What is a Good Bot Platform?

Bot platforms can be diverse.

There are many areas in the process where efficiency is a key factor and platforms can address one to many segments of the problem domain.

There are platforms for pre-development, mid-development and post-development.

The beauty is that the space is big enough for everyone to thrive. However for the developer, that means integrating multiple platforms and spending time to make sure all systems can talk to each other.

They are in a tandem, system integrator rather than platforms. This step can be frustrating and time consuming.

A good platform should take advantage of all the standards and best practices, giving developers the flexibility to integrate and mix-match between different services to solve their problem without platform being in the way.

Cross-Channel

Cross-Channel refers to your bot's ability to be deployed across different messaging platform.This is possible using a single code base (hence crossing to different platforms).

Cross-Channel support is not only a huge time saver (recreating development environment, bot, testing, deployment) but allows for quicker go-to-market possibilities. You can launch your bot across the different channels your customers are using and letting them choose how they would want to interact with you.

Conversations happen across many channels; Facebook messenger, Slack, WeChat etc. Why limit yourself to one subset of your users when you can maximize your impact and reach millions more? Developing on a platform that enables cross-channel deployment saves you time, effort and resources. Most people launch mobiles apps on both the Apple App Store and the Google Play Store. Why is it not the same for bots?

As a developer, you simply reuse your code. There is no need to incorporate additional flows since the logic remains the same.

Productivity & Performance

The biggest benefit in using a bot platform is the increased productivity of your development team and increased scalability of your bot. Not only can the bot platform power your process and hosting, they can help you improve and scale your bot.

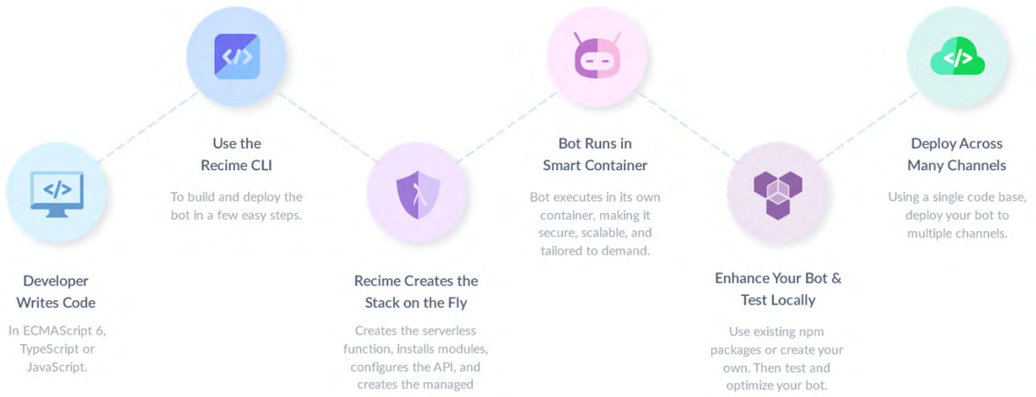
A Good Bot Platform Example

Recime.io (bitly.com/recime-dncmag) is a cloud bot infrastructure back-end and hosting platform (Bot-as-a-Service). Recime handles creating a container, server setup, wiring up corresponding SDKs, integrating NLP, adding API access tokens and hosting it all on the cloud for deploying on multiple channels and platform.

Recime is fully managed, giving developers the freedom to focus on their core product without the distraction of maintaining servers, hardware, or infrastructure (SaaS Model). The Recime experience provides services, tools, workflows, cross-platform enablement and support—all designed to enhance developer productivity.

A typical developer using a bot platform (e.g. Recime) has this type of workflow:

Looking at the original process flow, it is shortened into:



Summary

More and more companies are adding chatbots as part of their marketing strategy.

This new paradigm shift in creating communication channels powered by AI will drive future consumer interaction. From shopping to customer service, the world will be powered by bots (it is already happening).

Bots direct consumer to specific conversational flow, to pre-defined goal and data insights on what consumers are searching for. This predictive data can give you an edge over competitors. Some of the biggest brands in the world are already using bots.

In the end, it's about growth, productivity and customer success. Bot platforms can play a significant role by putting scale, cost, analytics, integration and time saving factors to your advantage.

Sign up for a free Recime account and start building your own bot! ■

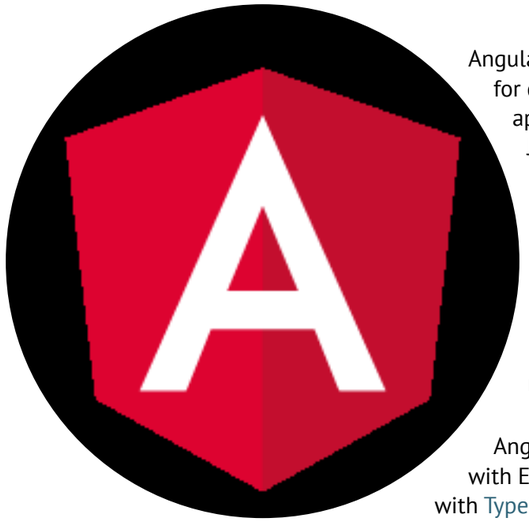
Mehfuz Hossain
Author

Mehfuz leads the team with a commanding knowledge of technologies and is constantly working to stay abreast of the changes happening in the software industry. As a software engineer for more than 10 years, Mehfuz created Recime to improve the way developers create the bot of tomorrow.

Thanks to Rahul Sahasrabuddhe for reviewing this article.

Keerti Kotaru

ANGULAR 4 DEVELOPMENT CHEAT SHEET



Angular is a JavaScript framework for developing mobile and web applications. It started as a client side JavaScript framework for writing better front-end applications that run in a web browser. Today, Angular leverages advancements made in modern web development to create web applications for desktop browsers, mobile web, mobile cross-platform, as well as native mobile applications.

Angular applications can be written with ES5 and/or ECMAScript 2015, along with TypeScript. TypeScript (bit.ly/dnc-ts-tut) is a popular choice for Angular applications as TypeScript is open- source, has powerful type-checking abilities, and amongst many other features, provides auto-completion, navigation and refactoring, which are very useful in large projects. Since TypeScript leverages ES6 as part of its core foundation, you feel you are very much using ES6 on steroids.

This article is a cheat sheet for using Angular with TypeScript. It's a good quick start guide by V Keerti Kotaru to get you going with Angular development.

Angular 4 Cheat Sheet

01. Components

Components are building blocks of an Angular application. A component is a combination of HTML template and a TypeScript (or a JavaScript) class. To create a component in Angular with TypeScript, create a class and decorate it with the Component decorator.

Consider the sample below:

```
import { Component } from '@angular/core';
@Component({
  selector: 'hello-ng-world',
  template: `<h1>Hello Angular world</h1>`
})
export class HelloWorld {
}
```

Component is imported from the core angular package. The component decorator allows specifying metadata for the given component. While there are many metadata fields that we can use, here are some important ones:

selector: is the name given to identify a component. In the sample we just saw, hello-ng-world is used to refer to the component in another template or HTML code.

template: is the markup for the given component. While the component is utilized, bindings with variables in the component class, styling and presentation logic are applied.

templateUrl: is the url to an external file containing a template for the view. For better code organization, template could be moved to a separate file and the path could be specified as a value for templateUrl.

styles: is used to specific styles for the given component. They are scoped to the component.

styleUrls: defines the CSS files containing the style for the component. We can specify one or more CSS files in an array. From these CSS files, classes and other styles could be applied in the template for the component.

02. Data-binding

String interpolation is an easy way to show data in an Angular application. Consider the following syntax in an Angular template to show the value of a variable.

```
`<h1>Hello {{title}} world</h1>`
```

`title` is the variable name. Notice single quotes (backtick `) around the string which is the ES6/ES2015 syntax for mixing variables in a string.

The complete code snippet for the component is as follows:

```
import { Component } from '@angular/core';
@Component({
```

```
  selector: ‘hello-ng-world’,
  template: `<h1>Hello {{title}} world</h1>`
})
export class HelloWorld {
title = ‘Angular 4’;
}
```

To show the value in a text field, use DOM property “*value*” wrapped in square brackets. As it uses a DOM attribute, it’s natural and easy to learn for anyone who is new to Angular. “title” is the name of the variable in the component.

```
<input type=”text” [value]=”title”>
```

Such a binding could be applied on any HTML attribute. Consider the following example where the title is used as a placeholder to the text field.

```
<input type=”text” [placeholder]=”title” >
```

03. Events

To bind a DOM event with a function in a component, use the following syntax. Wrap the DOM event in circular parenthesis, which invokes a function in the component.

```
<button (click)=”updateTime()”>Update Time</button>
```

3.1 Accepting user input

As the user keys-in values in text fields or makes selections on a dropdown or a radio button, values need to be updated on component/class variables.

We may use **events** to achieve this.
The following snippet updates value in a text field, to a variable on the component.

```
<!— Change event triggers function updateValue on the component -->
<input type=”text” (change)=”updateValue($event)”>
  updateValue(event: Event){
    // event.target.value has the value on the text field.
    // It is set to the label.
    this.label = event.target.value;
  }
```

Note: Considering it’s a basic example, the *event* parameter to the function above is of type *any*. For better type checking, it’s advisable to declare it of type *KeyboardEvent*.

3.2 Accepting user input, a better way

In the Accepting user input sample, $event is exposing a lot of information to the function/component. It encapsulates the complete DOM event triggered originally. However, all the function needs is the value of the text field.

Consider the following piece of code, which is a refined implementation of the same. Use template reference variable on the text field.

```
<input type=”text” #label1 (change)=”updateValue(label1.value)”>
```

Notice *updateValue* function on *change*, which accepts value field on label1 (template reference variable). The update function can now set the value to a class variable.

```
  updateValue(value: any){
    // It is set to the label.
    this.label = value;
  }
```

3.3 Banana in a box

From Angular 2 onwards, *two way data binding* is not implicit. Consider the following sample. As the user changes value in the text field, it’s doesn’t instantly update the title in h1.

```
<h1> {{title}} </h1>
<input type=”text” [value]=”title”>
```

Use the following syntax for two-way data binding. It combines value binding and event binding with the short form – [()]. On a lighter note, it’s called banana in a box.

```
<h1> {{title}} </h1>
<input type=”text” [(ngModel)]=”title” name=”title”>
```

04. ngModel and form fields

ngModel is not only useful for two-way data binding, but also with certain additional CSS classes indicating state of the form field.

Consider the following form fields - first name and last name. ngModel is set to fname and lname fields on the view model (the variable vm defined in the component).

```
    <input type=”text” [(ngModel)]=”vm.fname” name=”firstName” #fname required />
{{fname.className}} <br />
    <input type=”text” [(ngModel)]=”vm.lname” name=”lastName” #lname /> {{lname.
className}}
```

Please note, if ngModel is used within a form, it’s required to add a name attribute. The control is registered with the form (parent using the name attribute value. Not providing a name attribute will result in the following error.

If ngModel is used within a form tag, either the name attribute must be set or the form control must be defined as ‘standalone’ in ngModelOptions.

It adds the following CSS classes as and when the user starts to use the form input elements.

ng-untouched – The CSS class will be set on the form field as the page loads. The user hasn’t used the field and didn’t set keyboard focus on it yet.

ng-touched – The CSS class will be added on the form field as user sets focus by moving keyboard cursor or clicks on it. Once the user moves away from the field, this class is set.

ng-pristine – This class is set as long as the value on the field hasn't changed.

ng-dirty – This class is set while user modifies the value.

ng-valid – This class is set when all form field validations are satisfied, none failing. For example a required field has a value.

ng-invalid – This class is set when the form field validations are failing. For example, a required field doesn't have a value.

This feature allows customizing CSS class based on a scenario. For example, an invalid input field could have red background highlighting it for the user.

Define the following CSS for the given component with form elements.

```
.ng-invalid{
  background: orange;
}
```

Check out figure 1. It depicts two form fields with CSS classes, indicating state of the form field next to it.

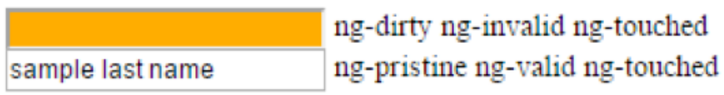


Figure 1: Input fields and CSS classes applied on it

05. ngModule

Create an Angular module by using the `ngModule` decorator function on a class.

A module helps package Angular artifacts like components, directives, pipes etc. It helps with ahead-of-time (AoT) compilation. A module exposes the components, directives and pipes to other Angular modules. It also allows specifying dependency on other Angular modules.

Consider the following code sample.

Note: *All components have to declared with an Angular module.*

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { MyAppComponent }  from './app.component';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ MyAppComponent ],
  bootstrap:    [ MyAppComponent ]
})
export class MyAppModule { }
```

Import *ngModule* from Angular core package. It is a decorator function that can be applied on a class. The following metadata is provided to create a module.

imports: Dependency on *BrowserModule* is specified with *imports*. Notice it's an array. Multiple modules' dependency could be specified with imports.

declarations: All components associated with the current module could be specified as values in *declarations* array.

bootstrap: Each Angular application needs to have a root module which will have a root component that is rendered on index.html. For the current module, MyAppComponent (imported from a relative path) is the first component to load.

06. Service

Create a service in an Angular application for any reusable piece of code. It could be accessing a server side service API, providing configuration information etc.

To create a service, import and decorate a class with *@injectable* (TypeScript) from *@angular/core module*. This allows angular to create an instance and inject the given service in a component or another service.

Consider the following hypothetical sample. It creates a *TimeService* for providing date time values in a consistent format across the application. The `getTime()` function could be called in a component within the application.

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

@Injectable()
export class TimeService {
  constructor() { }
  getTime(){
    return `${new Date().getHours()} : ${new Date().getMinutes()} : ${new Date().
    getSeconds()}`;
  }
}
```

6.1 Dependency Injection (DI)

In the service section, we have seen creating a service by decorating it with `Injectable()` function. It helps angular injector create an object of the service. It is required only if the service has other dependencies injected. However, it's a good practice to add `Injectable()` decorator on all services (even the ones without dependencies) to keep the code consistent.

With DI, Angular creates an instance of the service, which offloads object creation, allows implementing singleton easily, and makes the code conducive for unit testing.

6.2 Provide a service

To inject the service in a component, specify the service in a list of a providers. It could be done at the module level (to be accessible largely) or at component level (to limit the scope). The service is instantiated by Angular at this point.

Here's a component sample where the injected service instance is available to the component and all its child components.

```
import { Component } from '@angular/core';
import { TimeService } from './time.service';
@Component({
  selector: 'app-root',
  providers: [TimeService],
  template: `<div>Date: {{timeValue}}</div>,
})
export class SampleComponent {
  // Component definition
}
To complete the DI process, specify the service as a parameter property in the
constructor.
export class SampleComponent {
  timeValue: string = this.time.getTime(); // Template shows

  constructor(private time: TimeService){
  }
}
```

Refer to the TimeService implementation below,

```
import { Injectable } from '@angular/core';
@Injectable()
export class TimeService {
  constructor() { }
  getTime(){
    let dateObj: Date = new Date();
    return `${dateObj.getDay()}/${dateObj.getMonth()}/${dateObj.getFullYear()}`;
  }
}
```

6.3 Provide a service at module level

When the TimeService is provided at module level (instead of the component level), the service instance is available across the module (and application). It works like a Singleton across the application.

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

6.4 Alternative syntax to provide a service

In the above sample, provider statement is a short cut to the following,

```
// instead of providers: [TimeService] you may use the following,
  providers: [{provide: TimeService, useClass: TimeService}]
```

It allows using a specialized or alternative implementation of the class (TimeService in this example). The new class could be a derived class or the one with similar function signatures to the original class.


```
providers: [{provide: TimeService, useClass: AlternateTimeService}]
```

For the sake of an example, the *AlternateTimeService* provides date and time value. The original *TimeService* provided just the date value.

```
@Injectable()
export class AlternateTimeService extends TimeService {
  constructor() {
    super();
   }

  getTime(){
    let dateObj: Date = new Date();
    return `${dateObj.getDay()}/${dateObj.getMonth()}/${dateObj.getFullYear()}
                ${dateObj.getHours()}:${dateObj.getMinutes()}`;
  }
}
```

Note: Angular injector creates a new instance of the service when the following statement is used to provide a service:

```
[{provide: TimeService, useClass: AlternateTimeService}]
```

To rather use an existing instance of the service, use `useExisting` instead of `useClass`:

```
 providers: [AlternateTimeService, {provide: TimeService, useExisting:
AlternateTimeService}]
```

6.5 Provide an Interface and its implementation

It might be a good idea to provide an interface and implementation as a class or a value. Consider the following interface.

```
interface Time{
  getTime(): string
}
export default Time;

It could be implemented by TimeService.
@Injectable()
export class TimeService implements Time {
  constructor() { }

  getTime(){
    let dateObj: Date = new Date();
    return `${dateObj.getDay()}/${dateObj.getMonth()}/${dateObj.getFullYear()}`;
  }
}
```

At this point, we can't implement Time interface and TimeService class like the above sample.

The following piece of code does **not** work, because a TypeScript interface doesn't compile to any equivalent in JavaScript.

```
 providers: [{provide: Time, useClass: TimeService}] // Wrong
```

To make this work, import Inject (Decorator) and InjectionToken from @angular/core.

```
// create an object of InjectionToken that confines to interface Time
let Time_Service = new InjectionToken<Time>('Time_Service');

// Provide the injector token with interface implementation.
providers: [{provide: Time_Service, useClass: TimeService}]

// inject the token with @inject decorator
constructor(@Inject(Time_Service) ts,) {
  this.timeService = ts;
}

// We can now use this.timeService.getTime().
```

6.6 Provide a value

We may not always need a class to be provided as a service.

The following syntax allows for providing a JSON object. Notice JSON object has the function `getTime()`, which could be used in components.

```
providers: [{provide: TimeService, useValue: {
getTime: () => `${dateObj.getDay()} - ${dateObj.getMonth()} - ${dateObj.
getFullYear()}`
  }}]
```

Note: For an implementation similar to the section "*Provide an Interface and its implementation*", provide with *useValue* instead of *useClass*. The rest of the implementation stays the same.

```
providers: [provide: Time_Service, useValue: {
  getTime: () => 'A date value'
}}]
```

07. Directives

From Angular 2 onwards, directives are broadly categorized as following:

1. Components - Includes template. They are the primary building blocks of an Angular application. Refer to the Component section in the article.

2. Structural directives – A directive for managing layout. It is added as an attribute on the element and controls flow in DOM. Example NgFor, NgIf etc.

3. Attribute directives – Adds dynamic behavior and styling to an element in the template. Example NgStyle.

7.1 Ng-if... else

A structural directive to show a template conditionally. Consider the following sample. The *ngIf* directive guards against showing half a string *Time: [with no value*] when the title is empty. The *else* condition shows a template when no value is set for the *title*.

Notice the syntax here, the directive is prefixed with an asterisk.

```
<div *ngIf="title; else noTitle">
  Time: {{title}}
</div>

<ng-template #noTitle> Click on the button to see time. </ng-template>
```

As and when the *title* value is available, *Time: [value]* is shown. The *#noTitle* template hides, as it doesn't run *else*.

7.2 Ng-Template

Ng-Template is a structural directive that doesn't show by default. It helps group content under an element. By default, the template renders as a HTML comment.

The content is shown conditionally.

```
<div *ngIf="isTrue; then tmplWhenTrue else tmplWhenFalse"></div>
<ng-template #tmplWhenTrue >I show-up when isTrue is true. </ng-template>
<ng-template #tmplWhenFalse > I show-up when isTrue is false </ng-template>
```

Note: Instead of ng-if...else (as in *ng-if...else* section), for better code readability, we can use *if...then...else*. Here, the element shown when condition is true, is also moved in a template.

7.3 Ng-Container

Ng-container is another directive/component to group HTML elements.
We may group elements with a tag like `div` or `span`. However, in many applications, there could be default style applied on these elements. To be more predictable, Ng-Container is preferred. It groups elements, but doesn't render itself as a HTML tag.

Consider following sample,

```
// Consider value of title is Angular
Welcome <div *ngIf="title">to <i>the</i> {{title}} world.</div>
```

It is rendered as

> Welcome
> to *the* Angular world.

The resultant HTML is as below,

```
Welcome "
<!--bindings={
  "ng-reflect-ng-if": "Angular"
}-->
▼<div _ngcontent-c0>
  "to "
  <i _ngcontent-c0>the</i>
  " Angular world."
</div>
<br _ngcontent-c0>
```

It didn't render in one line due to the div. We may change the behavior with CSS. However, we may have a

styling applied for `div` by default. That might result in an unexpected styling to the string within the `div`.

Now instead, consider using ng-container. Refer to the snippet below.

```
Welcome <ng-container *ngIf=”title”>to <i>the</i> {{title}} world.</ng-container>
The result on the webpage is as following
```

> Welcome to *the* Angular world.

The resultant HTML is as below. Notice *ng-container* didn't show up as an element.

```
"

Welcome "
<!--bindings={
  "ng-reflect-ng-if": "Angular"
}-->
<!---->
"to "
<i _ngcontent-c0>the</i>
" Angular world."
<br _ngcontent-c0>
```

7.4 NgSwitch and NgSwitchCase

We can use **switch-case** statement in Angular templates. It's similar to switch..case statement in JavaScript.

Consider the following snippet. *isMetric* is a variable on the component. If its value is true, it will show Degree Celsicus as the label, else it will show Fahrenheit.

Notice *ngSwitch* is an attribute directive and *ngSwitchCase* is a structural directive.

```
<div [ngSwitch]=”isMetric”>
  <div *ngSwitchCase=”true”>Degree Celsius</div>
  <div *ngSwitchCase=”false”>Fahrenheit</div>
</div>
```

Please note, we may use *ngSwitchDefault* directive to show a default element when none of the values in the switch…case are true.

Considering *isMetric* is a boolean variable, the following code will result in the same output as the previous snippet.

```
<div [ngSwitch]=”isMetric”>
  <div *ngSwitchCase=”true”>Degree Celsius</div>
  <div *ngSwitchDefault>Fahrenheit</div>
</div>
```

7.5 Input decorator function

A class variable could be configured as an input to the directive. The value will be provided by component using the directive.

Consider the following code snippet. It is a component that shows login fields. (A component is a type of directive).

The Component invoking login component could set *showRegister* to true, resulting in showing a register button.

To make a class variable input, annotate it with the Input decorator function.

```
Import Input()
import { Component, OnInit, Input } from ‘@angular/core’;
```

Annotate with the decorator function:

```
  @Input() showRegister: boolean;
```

Use the input value in the component which is in a template in this example.

```
<div>
  <input type=”text” placeholder=”User Id” />
  <input type=”password” placeholder=”Password” />
  <span *ngIf=”showRegister”><button>Register</button></span>
  <button>Go</button>
</div>
```

While using the component, provide the input:

```
<login shouldShowRegister=”true”></login>
```

To use binding instead of providing the value directly, use the following syntax:

```
<app-login [shouldShowRegister]=”isRegisterVisible”></app-login>
```

We could provide a different name to the attribute than that of the variable. Consider the following snippet:

```
  @Input(“should-show-register”) showRegister: boolean;
```

Now, use the attribute `should-show-register` instead of the variable name `showRegister`.

```
<app-login should-show-register=”true”></app-login>
```

7.6 Output decorator function

Events emitted by a directive are output to the component (or a directive) using the given directive. In the login example, on clicking login, an event could be emitted with user id and password.

Consider the following snippet. Import *Output* decorator function and *EventEmitter*,

```
import { Component, OnInit, Input, Output, EventEmitter } from ‘@angular/core’;
```

Declare an Output event of type *EventEmitter*

```
  @Output() onLogin: EventEmitter<{userId: string, password: string}>;
```

Notice the generic type (anonymous) on EventEmitter for onLogin. It is expected to emit an object with user id and password.

Next, initialize the object.

```
constructor() {
  this.onLogin = new EventEmitter();
}
```

The component emits the event. In the sample, when user clicks on login, it emits the event with a next function.

Here's the template:

```
<button (click)="loginClicked(userId.value, password.value)">Go</button>
```

..and the Event handler:

```
loginClicked(userId, password){
  this.onLogin.next({userId, password});
}
```

While using the component, specify login handler for the onLogin output event.

```
<app-login (onLogin)="loginHandler($event)"></app-login>
```

Login handler receives user id and password from the login component

```
loginHandler(event){
  console.log(event);
  // Perform login action.
}

// Output: Object {userId: "sampleUser", password: "samplePassword"}
```

Similar to the input decorator, output decorator could specify an event name to be exposed for consuming component/directive. It need not be the variable name. Consider the following snippet.

```
@Output("login") onLogin: EventEmitter<{userId: string, password: string}>;
```

..while using the component:

```
<app-login (login)="loginHandler($event)"></app-login>
```

08. Change Detection Strategy

Angular propagates changes top-down, from parent components to child components.

Each component in Angular has an equivalent change detection class. As the component model is updated, it compares previous and new values. Then changes to the model are updated in the DOM.

For component inputs like number and string, the variables are immutable. As the value changes and the change detection class flags the change, the DOM is updated.

For object types, the comparison could be one of the following,

Shallow Comparison: Compare object references. If one or more fields in the object are updated, object reference doesn't change. Shallow check will not notice the change. However, this approach works best for immutable object types. That is, objects cannot be modified. Changes need to be handled by creating a new instance of the object.

Deep Comparison: Iterate through each field on the object and compare with previous value. This identifies change in the mutable object as well. That is, if one of the field on the object is updated, the change is noticed.

8.1 Change detection strategy for a component

On a component, we can annotate with one of the two change detection strategies.

ChangeDetectionStrategy.Default:

As the name indicates, it's the default strategy when nothing is explicitly annotated on a component.

With this strategy, if the component accepts an object type as an input, it performs deep comparison every time there is a change. That is, if one of the fields have changed, it will iterate through all the fields and identify the change. It will then update the DOM.

Consider the following sample:

```
import { Component, OnInit, Input, ChangeDetectionStrategy } from '@angular/core';

@Component({
  selector: 'app-child',
  template: '  <h2>{{values.title}}</h2> <h2>{{values.description}}</h2>',
  styleUrls: ['./child.component.css'],
  changeDetection: ChangeDetectionStrategy.Default
})
export class ChildComponent implements OnInit {

// Input is an object type.
  @Input() values: {
    title: string;
    description: string;
  }
  constructor() { }

  ngOnInit() {}
}
```

Notice, change detection strategy is mentioned as *ChangeDetectionStrategy.Default*, it is the value set by default in any component. *Input* to the component is an object type with two fields *title* and *description*.

Here's the snippet from parent component template.

```
<app-child [values]="values"></app-child>
```

An event handler in the parent component is mentioned below. This is triggered, possibly by a user action or other events.

```
updateValues(param1, param2){
    this.values.title = param1;
    this.values.description = param2;
  }
```

Notice, we are updating the values object. The Object reference doesn't change. However the child component updates DOM as the strategy performs deep comparison.

ChangeDetectionStrategy.OnPush:

The default strategy is effective for identifying changes. However, it's not performant as it has to loop through a complete object to identify change. For better performance with change detection, consider using *OnPush* strategy. It performs shallow comparison of input object with the previous object. That is, it compares only the object reference.

The code snippet we just saw will not work with *OnPush* strategy. In the sample, the reference doesn't change as we modify values on the object. The Child component will not update the DOM.

When we change strategy to *OnPush* on a component, the input object needs to be immutable. We should create a new instance of input object, every time there is a change. This changes object reference and hence the comparison identifies the change.

Consider the following snippet for handling change event in the parent.

```
updateValues(param1, param2){
  this.values = { // create a new object for each change.
    title: param1,
    description: param2
  }
}
```

Note: Even though the above solution for *updateValues* event handler might work okay with the *OnPush* strategy, it's advisable to use immutable.js implementation for enforcing immutability with objects or observable objects using RxJS or any other observable library.

09. Transclusion in Angular

In the above two sections (input decorator and output decorator), we have seen how to use component attributes for input and output.

How about accessing content within the element, between begin and end tags?

AngularJS 1.x had transclusion in directives that allowed content within the element to render as part of the directive. For example, an element <blue></blue> can have elements, expressions and text within the element. The component will apply blue background color and show the content.

```
<blue>sample text</blue>
```

In the blue component's template, use ng-content to access content within the element.

Consider the following. The CSS class blue applies styling.

```
<divclass="blue">
  <ng-content>
</ng-content></div>
```


ng-content shows *sample text* from above example. It may contain more elements, data binding expressions etc.

10. Using observables in Angular templates

Observable data by definition, may not be available while rendering the template. The *ngIf directive could be used to conditionally render a section. Consider the following sample.

```
<div *ngIf="asyncData | async; else loading; let title">
  Title: {{title}}
</div>
<ng-template #loading> Loading... </ng-template>
```

Notice the async pipe. The sample above checks for asyncData observable to return with data. When observable doesn't have data, it renders the template *loading*. When the observable returns data, the value is set on a variable title. The async pipe works the same way with promises as well.

NgFor

Use *ngFor directive to iterate through an array or an observable. The following code iterates through colors array and each item in the array is referred to as a new variable color.

```
<ul *ngFor="let color of colors">
  <li>{{color}}</li>
</ul>
/* component declares array ascolors= ["red", "blue", "green", "yellow",
"violet"];*/
```

Note: Use *async pipe* if colors is an observable or a promise.

11. Strict Null Check

TypeScript 4 introduced strict null check for better type checking. TypeScript (& JavaScript) have special types namely *null* and *undefined*.
With strict type check enabled for an Angular application, null or undefined cannot be assigned to a variable unless they are of that type. Consider the following sample:

```
varfirstName:string, lastName:string ;
//returns an error, Type 'null' is not assignable to type 'string'.
firstName=null;
// returns an error, Type 'undefined' is not assignable to type 'string'.
lastName=undefined;
```

Now explicitly specify null and undefined types. Revisit variable declaration as following:

```
varfirstName:string|null, lastName:string|undefined ;
//This will work
firstName=null;
lastName=undefined;
```

By default, strictNullCheck is disabled. To enable strictNullCheck edit tsconfig.json. Add "strictNullChecks": true in compilerOptions

```
{
  "compileOnSave": false,
  "compilerOptions": {
    "outDir": "./dist/out-tsc",
    "baseUrl": "src",
    "sourceMap": true,
    "declaration": false,
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "strictNullChecks": true,
    "target": "es5",
    "typeRoots": [
      "node_modules/@types"
    ],
```

Figure 2: tsconfig.json

12. HTTP API calls and Observables

Use the built-in Http service from '@angular/http' module to make server side API calls. Inject the Http service into a component or another service. Consider this snippet which makes a GET call to a Wikipedia URL. Here http is an object of Http in @angular/http

```
this.http
  .get("https://dinosaur-facts.firebaseio.com/dinosaurs.json")
```

The get() function returns an observable. It helps return data asynchronously to a callback function. Unlike promise, an observable supports a stream of data. It's not closed as soon as data is returned. Moreover it can be returned till the observable is explicitly closed.

Observables support multiple operators or chaining of operators. Consider the map operator below which extracts JSON out of the http response.

```
this.http
  .get("http://localhost:3000/dino")
  .map((response) => response.json())
```

Note: Shown here is one of the ways to import map operator.

```
import 'rxjs/add/operator/map'; // Preferred way to import
```

Otherwise, we may use the following import statement for the entire rxjs library.

```
import 'rxjs';
```

However, it still returns an observable. An observable has a subscribe function, which accepts three callbacks.

a) Success callback that has data input
b) Error callback that has error input
c) Complete callback, which is called while finishing with the observable.

Consider the following snippet,

```
this.http
  .get("https://dinosaur-facts.firebaseio.com/dinosaurs.json")
  .map((response) => response.json())
  .subscribe((data) => console.log(data), // success
  (error) => console.error(error), // failure
  () => console.info("done")); // done
```

```
                                                    form-sample.component.ts:20
▼ Object {bruhathkayosaurus: Object, lambeosaurus: Object, linhenykus: Object,
  pterodactyl: Object, stegosaurus: Object…}
  ▶ bruhathkayosaurus: Object
  ▶ lambeosaurus: Object
  ▶ linhenykus: Object
  ▶ pterodactyl: Object
  ▶ stegosaurus: Object
  ▶ triceratops: Object
  ▶ __proto__: Object
done                                                form-sample.component.ts:22
```

Figure 3: console output of observable.

You may set the observable result to a class/component variable and display it in the template. A better option would be to use async pipe in the template.
Here's a sample:

```
dino: Observable<any>; // class level variable declaration

// in the given function, where the API call run set returned observable to the //
class variable

this.dino = this.http // Set returned observable to a class variable
  .get("https://dinosaur-facts.firebaseio.com/dinosaurs.json")
  .map((response) => response.json());
```

In the template use async pipe. When the dino object is available, use fields on the object.

```
<div>{{ (dino |async)?.bruhathkayosaurus.appeared }}</div>
```

13. Take me back to Promises

We may use an RxJS operator to convert the *observable* to a *promise*. If an API currently returns a promise, for backward compatibility, it's a useful operator.

Import *toPromise* operator:

```
import 'rxjs/add/operator/toPromise';
```

Use *toPromise* on an observable:

```
getData(): Promise<any>{
  return this.http
  .get("https://dinosaur-facts.firebaseio.com/dinosaurs.json")
  .map((response) => response.json())
  .toPromise();
}
```

Use data returned on the success callback. Set it to a class variable to be used in the template.

```
this.getData()
  .then((data) =>  this.dino = data)
  .catch((error) => console.error(error));
```

Review the following template:

```
<div *ngIf=”dino”>
  <div>Bruhathkayosaurus appeared {{ dino.bruhathkayosaurus.appeared }} years ago
  </div>
</div>
```

14. Router

Routing makes it possible to build an SPA (Single Page Application). Using the router configuration, components are mapped to a URL.

To get started with an Angular application that supports routing using Angular CLI, run the following command.

```
ng new sample-application --routing
```

To add a module with routing to an existing application using Angular CLI, run the following command.

```
ng g module my-router-module --routing
```

14.1 Router Outlet

Components at the route (URL) are rendered below router outlet. In general, RouterOutlet is placed in the root component for routing to work.

```
<router-outlet></router-outlet>
```

14.2 Route configuration

Create a routes object of type Routes (in@angular/router package) which is an array of routes JSON objects. Each object may have one or more of the following fields.

path: Specifies path to match

component: At the given path, the given component will load below router-outlet

redirectTo: Redirects to the given path, when path matches. For example, it could redirect to home, when no path is provided.

pathMatch: It allows configuring path match strategy. When the given value is *full*, the complete path needs to match. Whereas *prefix* allows matching initial string. Prefix is the default value.

Consider the following route configuration for sample:

```
const routes: Routes = [
  {
    path: ‘home’,
    component: Sample1Component,
  },
  {
    path: ‘second2’,
    component: Sample2Component
  },
  {
    path: ‘’,
    redirectTo: ‘/home’,
    pathMatch: ‘full’
  }
];
```

14.3 Child Routes

For configuring child routes, use children within the route object. The child component shows-up at the router-outlet in the given component.

The following sample renders Sample1Component. It has a router-outlet in the template. Sample2Component renders below it.

```
const routes: Routes = [
  {
    path: ‘home’,
    component: Sample1Component,
    children: [
      {
        path: ‘second’,
        component: Sample2Component
      }
    ]
  }… // rest of the configuration.

// Template for Sample1Component

<div>
  sample-1 works!
  <router-outlet></router-outlet> <!--Sample2Component renders below it -->
</div>
```

14.4 Params

Data can be exchanged between routes using URL parameters.

Configure variable in the route: In the sample below, the *details* route expects an id as a parameter in the URL. Example http://sample.com/details/10

```
{
  path: details/:id,
  component: DetailsComponent
}
```

Read value from the URL: Import ActivatedRoute from @angular/router. Inject ActivatedRoute and access params. It’s an observable to read value from the URL as seen in the sample:

```
// inject activatedRoute
constructor(private activeRoute: ActivatedRoute) { }

// Read value from the observable
this.activeRoute
```

```
.params
.subscribe((data) => console.log(data[id]));
```

Conclusion

Angular as a JavaScript framework has been evolving fast. From being an MV* framework in AngularJS 1.x, it has become a framework that approaches UI development with reusable components.

With Angular 2 and above (current version being 4.x) performance has improved and new features are continuously added. The newer versions are referred to as just Angular, instead of qualifying it with a version number.

Angular with TypeScript has multitude of features that make development in JavaScript faster and easier. The article described some of the features to begin developing with Angular.

Happy coding with the super heroic framework, Angular!

References

Angular documentation – https://angular.io/docs

Change Detection Strategy with Angular - https://angular-2-training-book.rangle.io/handout/change-detection/change_detector_classes.html

Dinosaur data - https://dinosaur-facts.firebaseio.com/dinosaurs.json (Firebase API for Dinosaur data)

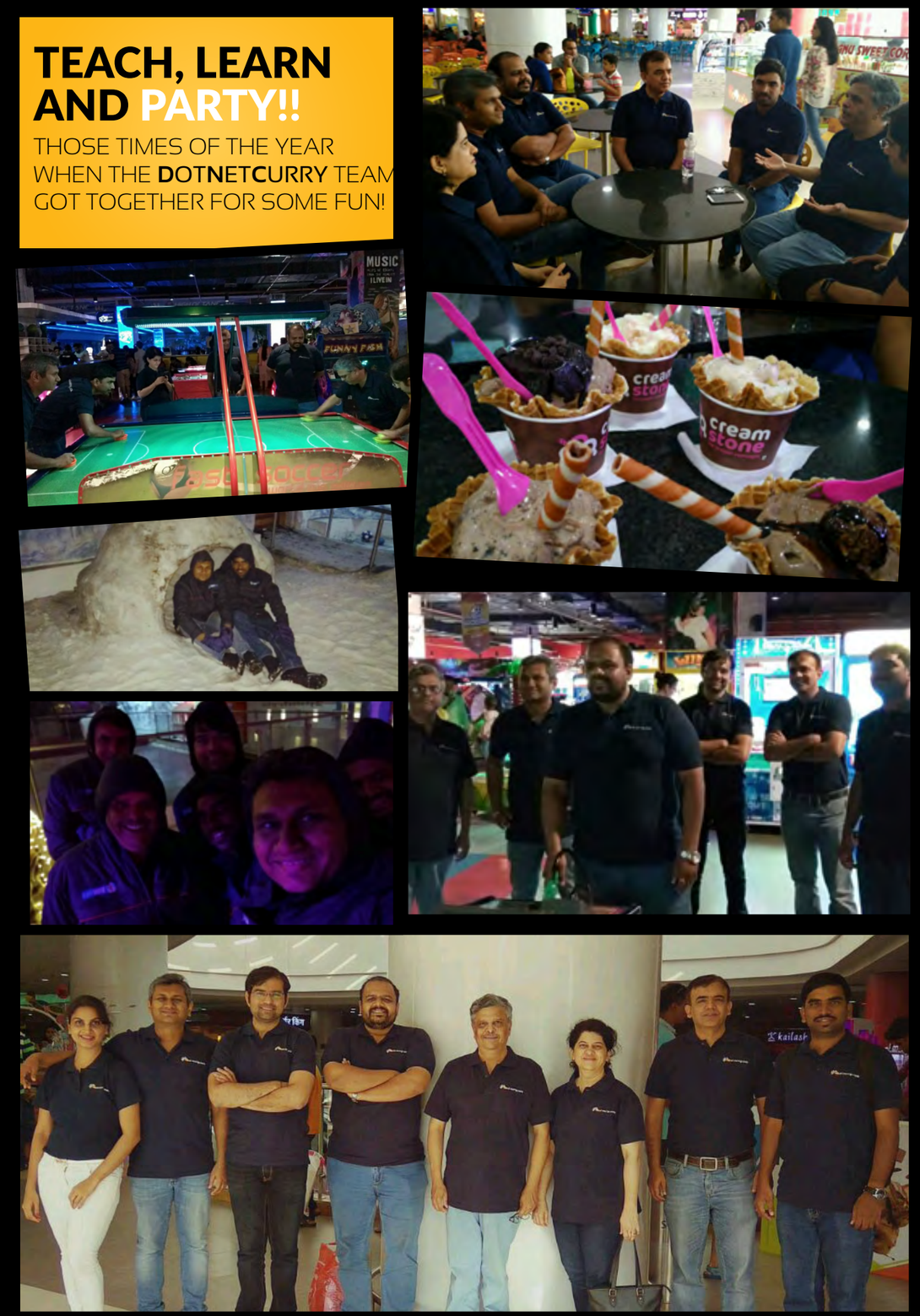
Microsoft MVP Most Valuable Professional

Keerti Kotaru

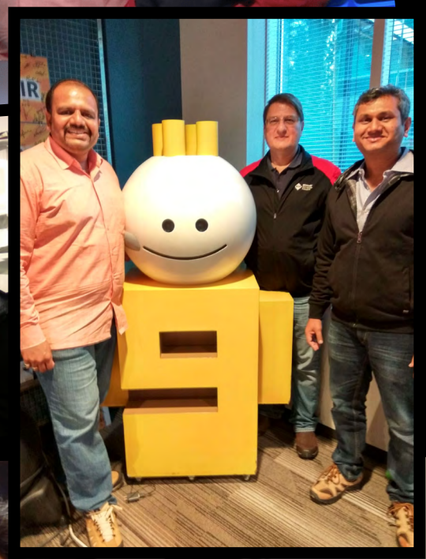
Author

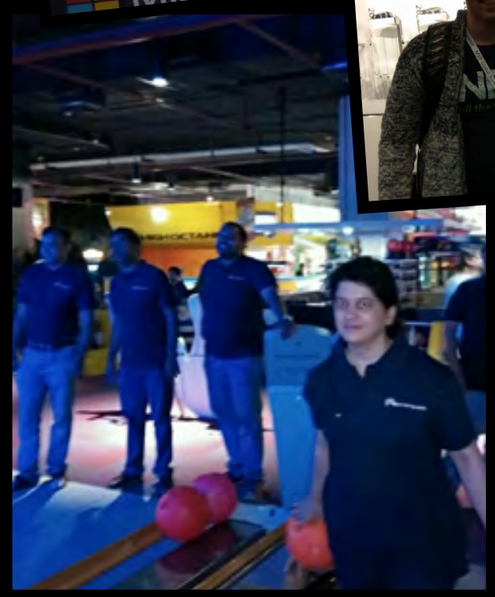
V Keerti Kotaru has been working on web applications for over 15 years now. He started his career as an ASP.Net, C# developer. Recently, he has been designing and developing web and mobile apps using JavaScript technologies. Keerti is also a Microsoft MVP, author of a book titled 'Material Design Implementation 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 and Suprotim Agarwal for reviewing this article.



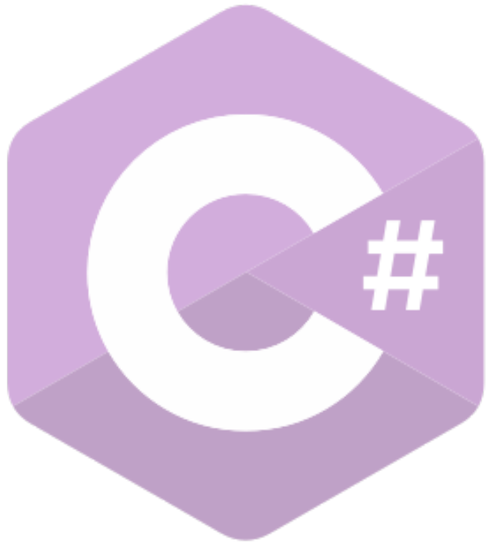
DNC Magazine raises a toast to its terrifically tireless, and spectacular authors who help millions to learn, prepare and stay ahead of the curve. Thank you!!!





Without YOU, this magazine would not exist! On behalf of the entire DotNetCurry (DNC) Magazine team and authors, Thank You for reading us!

Yacoub Massad



Using GENERICS IN C#

to Improve Application Maintainability

This article describes how we can use generics in C# to make our software more resilient to data-related changes.

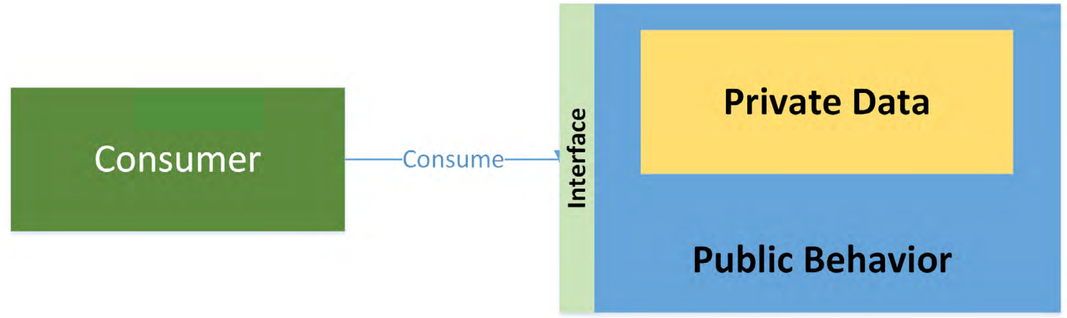
Introduction

In a previous article, Data and Encapsulation in complex C# applications, I talked about two ways to treat runtime data. More specifically, I talked about *data encapsulation* and *behavior-only encapsulation*.

In a nutshell, encapsulating data means that we have units (e.g. classes) that contain data in a manner that:

- prevents consumers from accessing it directly (e.g. by putting it in private fields),

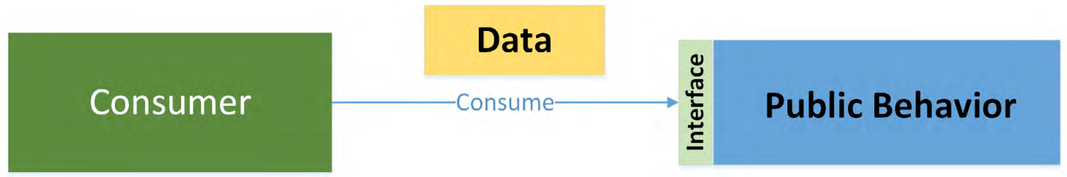
- allows consumers to use public methods that interact with it in an indirect way.



A different approach is to separate data and behavior into separate units.

Data units contain only data that is accessible directly, and behavior units contain only behavior. Such behavior units are similar to functions and procedures in functional and procedural programming, respectively.

Behavior encapsulation means that consumers of behavior units consume them indirectly via interfaces and therefore they don't know and don't care how they are implemented.



When we encapsulate data, we can change the internal representation of the data without affecting the consumers of the unit encapsulating the data.

On the other hand, when we don't encapsulate data, the behavior units will access data (received as method parameters for example) directly and therefore they have a hard dependency on the internal representation of the data.

Therefore, when doing behavior-only encapsulation, it is harder to change the internal representation of data.

In this article, I am going to discuss **how we can use generics in C# to make it easier to handle data-related change requests in software applications when doing behavior-only encapsulation.**

Note: *this article is not an introduction to generics in C#. I assume that the reader is already familiar with generics. For more information about generics, consider this guide from Microsoft.*

Data-related software changes

When maintaining an application, the development team receives many requirements that involve changing behavior with little or no changes to data structure/representation.

For example, one requirement might ask that we save documents to the database instead of the file system. Such requirements can be mostly met by creating new behavior units that contain the new behavior and then composing them correctly in the Composition Root.

On other occasions, however, requirements require our data objects to change or to support new data objects.

For example, in an application that processes (e.g. translates, prints, etc.) plain text documents, we might receive a requirement to support PDF documents or documents in Microsoft Word format.

Consider the following interface:

```
public interface ITextDocumentTranslator
{
  TextDocument Translate(TextDocument document, Language language);
}
public class TextDocument
{
  public string Identifier { get; }
  public string Content { get; }

  public TextDocument(string identifier, string content)
  {
    Identifier = identifier;
    Content = content;
  }
}
```

The `ITextDocumentTranslator` interface abstracts the process of translating a text-plain document to a different language. The `TextDocument` is a simple data object that represents a text document.

The `ITextDocumentTranslator` interface and its implementations have a hard dependency on `TextDocument` . They access the internals of this class (e.g. the Content property) freely. In a large application, we can expect to find tens of classes/interfaces that have a hard dependency on `TextDocument` .

In order to support PDF documents, we can create new interfaces and classes that deal with a `PdfDocument` data object. For instance, we can create a `IPdfDocumentTranslator` interface that looks like this:

```
public interface IPdfDocumentTranslator
{
    PdfDocument Translate(PdfDocument document, Language language);
}
```

The implementation of this interface would know how to translate a `PdfDocument` to a different language.

If the application originally had *fifty* classes and ten interfaces to deal with plain text documents, we would expect to have a similar number of classes and interfaces for also dealing with PDF documents.

A portion of the code in the new fifty (or so) classes is going to be very specific to PDF documents. After all, PDF documents have different features than plain text documents and processing them is going to be different than processing plain text documents.

On the other hand, another portion of the code in the new classes is going to be very similar to the old code. This is true because both sets of code are about processing documents.

Consider the following example:

```
public class TextDocumentProcessor : ITextDocumentProcessor
{
  private readonly ITextDocumentTranslator translator;
  private readonly ITextDocumentStore store;

  //..

  public void Process(TextDocument document)
  {
    var englishVersion = translator.Translate(document, Language.English);

    store.StoreDocument(englishVersion);
  }
}
```

This class is an **orchestrator**.

It receives a text document, invokes a translator dependency to translate it to English and then it invokes a store dependency to store the English version of the document. All it does is orchestration of operations; it doesn't know how the document is translated or where it is going to be stored.

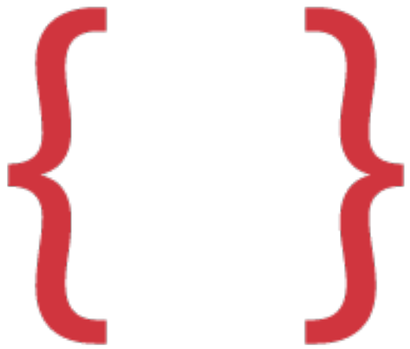
This orchestrator is **data-independent.**

Although it references the "document" parameter, it only uses it to pass it to the other dependencies. It has no dependency on the **internals** of the TextDocument data object.

If we create a similar class for PDF documents, it is going to be exactly the same as this one. This is clearly duplicate code:

```
public class PdfDocumentProcessor : IPdfDocumentProcessor
{
  private readonly IPdfDocumentTranslator translator;
  private readonly IPdfDocumentStore store;
```

```
  //..
  public void Process(PdfDocument document)
  {
    var englishVersion = translator.Translate(document, Language.English);
    store.StoreDocument(englishVersion);
  }
}
```



Generics in C#

Using generics to remove duplication

Instead of creating duplicate code, we can refactor the original code to deal with a generic document type. We do this with both classes and interfaces like this:

```
public interface IDocumentProcessor<TDocument>
{
  void Process(TDocument document);
}

public interface IDocumentTranslator<TDocument>
{
  TDocument Translate(TDocument document, Language language);
}

public interface IDocumentStore<TDocument>
{
  void StoreDocument(TDocument document);
}

public class DocumentProcessor<TDocument> : IDocumentProcessor<TDocument>
{
  private readonly IDocumentTranslator<TDocument> translator;
  private readonly IDocumentStore<TDocument> store;

  //..

  public void Process(TDocument document)
  {
    var englishVersion = translator.Translate(document, Language.English);

    store.StoreDocument(englishVersion);
  }
}
```

For this particular example, we have reduced the number of interfaces from six to three.

Also, now we have a single generic `DocumentProcessor<TDocument>` class instead of two classes. Because the orchestration is data-independent, it does not care about the internals of the document object and thus we were able to convert it into a generic class where the document type is generic.

For the plain text part of the application, we can construct an instance of DocumentProcessor<TDocument> like this:

```
var textDocumentProcessor =
  new DocumentProcessor<TextDocument>(
    new TextDocumentTranslator(...),
    new TextDocumentStore(...));
```

..and for the PDF part of the application, we can construct an instance of `DocumentProcessor<PdfDocument>` like this:

```
var pdfDocumentProcessor =
  new DocumentProcessor<PdfDocument>(
    new PdfDocumentTranslator(...),
    new PdfDocumentStore(...));
```

The TextDocumentTranslator, TextDocumentStore, PdfDocumentTranslator, PdfDocumentStore are non-generic classes that implement the generic interfaces for specific TDocument.

Here is one example:

```
public class TextDocumentStore : IDocumentStore<TextDocument>
{
  private readonly string connectionString;

  //..
  public void StoreDocument(TextDocument document)
  {
    using (var context = new DataContext(connectionString))
    {
      context.Documents.Add(
      new StoredDocument
      {
        Type = StoredDocumentType.PlainText,
        Identifier = document.Identifier,
        BinaryContent = Encoding.UTF8.GetBytes(document.Content),
      });
```

```
      context.SaveChanges();
    }
  }
}
```

This class is non-generic, and it implements the `IDocumentStore<TextDocument>` interface. It deals specifically with the `TextDocument` class. It accesses the Identifier and Content properties of the `TextDocument` data object.

Therefore, the class is **data-dependent**.

This class uses entity framework to store the document to the database. The `DataContext` class is a class derived from entity framework's `DbContext` class to enable accessing the database. See this article from Microsoft for more details.

Here is another one:

```
public class PdfDocumentStore : IDocumentStore<PdfDocument>
{
  private readonly string connectionString;

  //..
  public void StoreDocument(PdfDocument document)
  {
    using (var context = new DataContext(connectionString))
    {
      context.Documents.Add(
      new StoredDocument
      {
        Type = StoredDocumentType.Pdf,
        Identifier = document.Identifier,
        BinaryContent = SerializeDocument(document)
      });

      context.SaveChanges();
    }
  }

  private byte[] SerializeDocument(PdfDocument document)
  {
    //...
  }
}
```

This class is also non-generic, and it implements the `IDocumentStore<PdfDocument>` interface. Notice how it accesses specific properties from the `PdfDocument` data object. The `SerializeDocument` method accesses the internals of the `PdfDocument` object and somehow serializes the content of the PDF document into a byte array to store it in the database.

The exact details of how this is done is not relevant to the subject of this article.

Maximizing data-independency

We have seen in the previous sections that it is **data-independency** that allows us to make behavior units generic and therefore remove duplication.

A logical deduction is, the more the data-independent behavior units we have, the more the amount of code duplication we can remove.

But how can we do this?

Let's consider the `TextDocumentStore` and `PdfDocumentStore` units. These two units are data-dependent, each on its specific document type.

These two classes contain some shared logic. For example,
- both create an instance of the `DataContext` class passing a connection string,
- both create a new instance of the `StoredDocument` class and add it to the Documents DbSet,
- both invoke `SaveChanges` to commit the changes to the database, and
- both manage the connection by using the "using" statement.

However this shared logic is **data-independent**.

We can extract this shared logic into a data-independent class that replaces both the `TextDocumentStore` and the `PdfDocumentStore` classes, and then create two data-dependent classes just to extract the data specific to the two data types:

```
public interface IContentExtractor<TDocument>
{
  IdAndContent Extract(TDocument document);
}

public class DocumentStore<TDocument> : IDocumentStore<TDocument>
{
  private readonly IContentExtractor<TDocument> contentExtractor;
  private readonly string connectionString;
  private readonly StoredDocumentType documentType;

  //..

  public void StoreDocument(TDocument document)
  {
    var content = contentExtractor.Extract(document);

    using (var context = new DataContext(connectionString))
    {
       context.Documents.Add(
       new StoredDocument
       {
         Type = documentType,
         Identifier = content.Identifier,
         BinaryContent = content.Content
       });
```

```
      context.SaveChanges();
    }
  }
}

public class PdfContentExtractor : IContentExtractor<PdfDocument>
{
  public IdAndContent Extract(PdfDocument document)
  {
    return new IdAndContent(document.Identifier, SerializeDocument(document));
  }

  private byte[] SerializeDocument(PdfDocument document)
  {
    //Serialize Pdf document here
  }
}

public class TextContentExtractor : IContentExtractor<TextDocument>
{
  public IdAndContent Extract(TextDocument document)
  {
    return new IdAndContent(
      document.Identifier,
      Encoding.UTF8.GetBytes(document.Content));
  }
}
```

The amount of duplication has decreased.

The `DocumentStore` class is now a generic data-independent class that contains the logic to write data to the database.

When it needs data-type specific data, it uses the `IContentExtractor<TDocument>` dependency to obtain such data. For each data-type, we have a data-dependent implementation of this interface to extract the data.

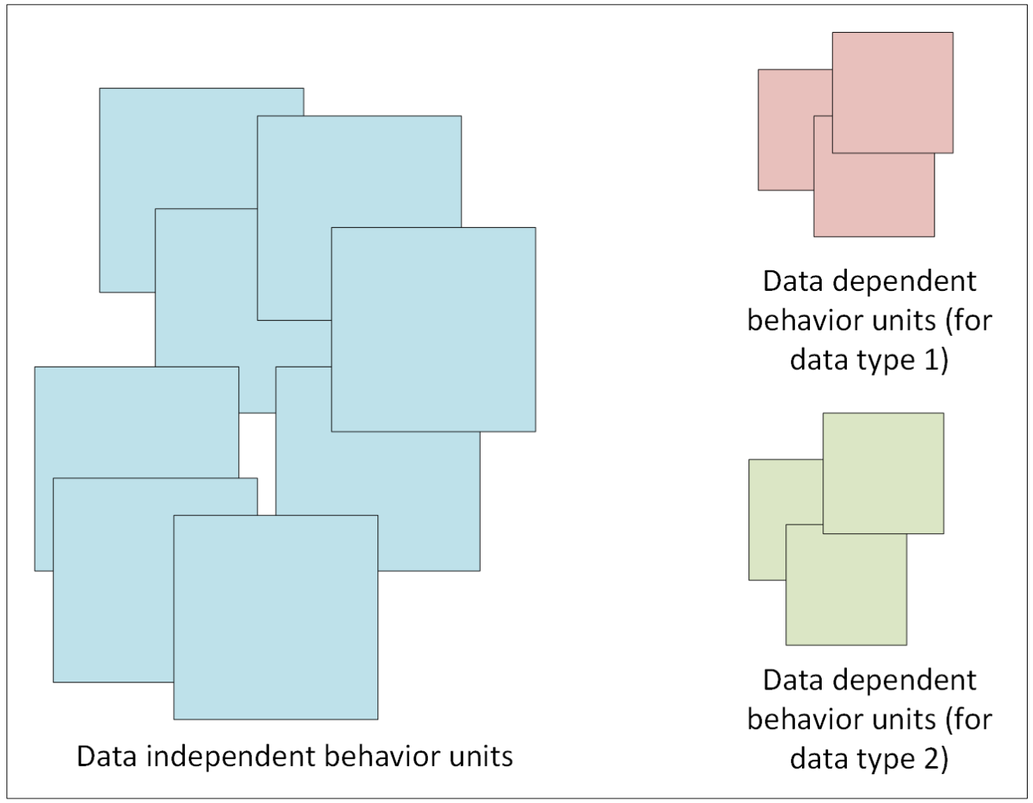
Here is how the Composition Root looks like now:

```
var textDocumentProcessor =
  new DocumentProcessor<TextDocument>(
    new TextDocumentTranslator(),
    new DocumentStore<TextDocument>(
      new TextContentExtractor(),
      connectionString,
      StoredDocumentType.PlainText));

var pdfDocumentProcessor =
  new DocumentProcessor<PdfDocument>(
    new PdfDocumentTranslator(),
    new DocumentStore<PdfDocument>(
      new PdfContentExtractor(),
      connectionString,
      StoredDocumentType.Pdf));
```

If the translation code in `TextDocumentTranslator` and `PdfDocumentTranslator` has shared logic, we can do a similar refactoring to move the shared data-independent part into generic data-independent classes, and move the data-dependent parts into their own classes.

After we are done with refactoring, data-independent code will be in generic data-independent classes, and for each data type, we will have smaller non-generic data-dependent classes to do the things that are specific to each data type.



Supporting changes within the context of a single data type

Separation of data-independent behavior and data-dependent behavior also has benefits within the context of a single data type.

Many a times, we are required to change the internal structure of our data types.

For example, we might decide to change the type of the `Content` property in the `TextDocument` data class from a simple string to a string array representing the text in each paragraph of the document. In this case, we will have to modify many of the data-dependent units because they depend on the old representation of the `Content` property.

When we separate data-independent behavior and data-dependent behavior into different units, we minimize the number and size of data-dependent units, and therefore it becomes easier to make changes to the internal structure of existing data types.

Moving data-dependent logic into the data objects

The PdfContentExtractor and TextContentExtractor classes from the previous example contain logic that is specific to PDF and plain text documents respectively.

One might be tempted to move such logic into the `PdfDocument` and `TextDocument` classes respectively. This is especially true when such logic is simple.

Here is how the relevant code would look like:

```
public interface IDocument
{
  string GetIdentifier();
  byte[] GetBinaryContent();
  StoredDocumentType GetTypeForStore();
}

public class TextDocument : IDocument
{
  public string Identifier { get; }
  public string Content { get; }

  //..

  public string GetIdentifier() => Identifier;

  public byte[] GetBinaryContent() => Encoding.UTF8.GetBytes(Content);

  public StoredDocumentType GetTypeForStore() => StoredDocumentType.PlainText;
}

public class PdfDocument : IDocument
{
  public string Identifier { get; }
  //..

  public string GetIdentifier() => Identifier;

  public byte[] GetBinaryContent()
  {
    //Serialize Pdf document here
  }

  public StoredDocumentType GetTypeForStore() => StoredDocumentType.Pdf;

}

public class DocumentStore<TDocument> : IDocumentStore<TDocument>
where TDocument: IDocument
{
  private readonly string connectionString;

  //..
  public DocumentStore(string connectionString)
  {
    this.connectionString = connectionString;
  }

  public void StoreDocument(TDocument document)
  {
    using (var context = new DataContext(connectionString))
    {
      context.Documents.Add(
        new StoredDocument
        {
          Type = document.GetTypeForStore(),
          Identifier = document.GetIdentifier(),
          BinaryContent = document.GetBinaryContent()
        });

      context.SaveChanges();
    }
  }
}
```

The `DocumentStore` class in this case does not need a dependency on `IContentExtractor<TDocument>` because the data dependent logic was moved to the document object itself. The DocumentStore can invoke methods via the `IDocument` interface to get the data-type specific data.

In this particular example, since it is very simple, such an approach is OK.

However, as I discussed in the Data and Encapsulation in complex C# applications article, this moves us towards data encapsulation, and in complex applications, this raises issues that we need to consider.

You aren't gonna need it (The YAGNI principle)

Should we always separate data-independent and data-dependent logic into different classes?

Should we make our data-independent classes generic from the start?

Well, it depends.

But in most of the cases, we don't have to.

In most of the cases, applications start small, and then they evolve with time. It is important to first concentrate efforts on meeting the requirements we have at hand. We can always refactor later to meet new requirements.

In a document processing application that processes plain text documents only, we can start normally without making the document type generic in the classes that deal with documents.

Later, when we need to introduce different documents types, we can refactor existing classes/interfaces to become generic, and we can also refactor to separate data-independent behavior and data-dependent behavior into different classes.

If we follow the SOLID principles (the Single Responsibility Principle in particular), chances are that separation of data-independent and data-dependent behavior, is already high. Also, refactoring in this case would be a lot easier compared to when our classes are very long.

Conclusion:

In this article, I discussed some data-related software changes and how generics can help us with them. One particular type of change is supporting new data types.

In many cases, the main application/domain logic doesn't change much when we add new data types. If we can separate the logic that is data-independent from the logic that is data-dependent, we can reuse a lot of code when we add new data types. Also, such separation helps us with changing the internal structure of data types by minimizing the number and size of behavior units that need to change as a result of such changes.

Generics in C# allow us to create data-independent behavior units that can be used for different data types

Yacoub Massad

Author

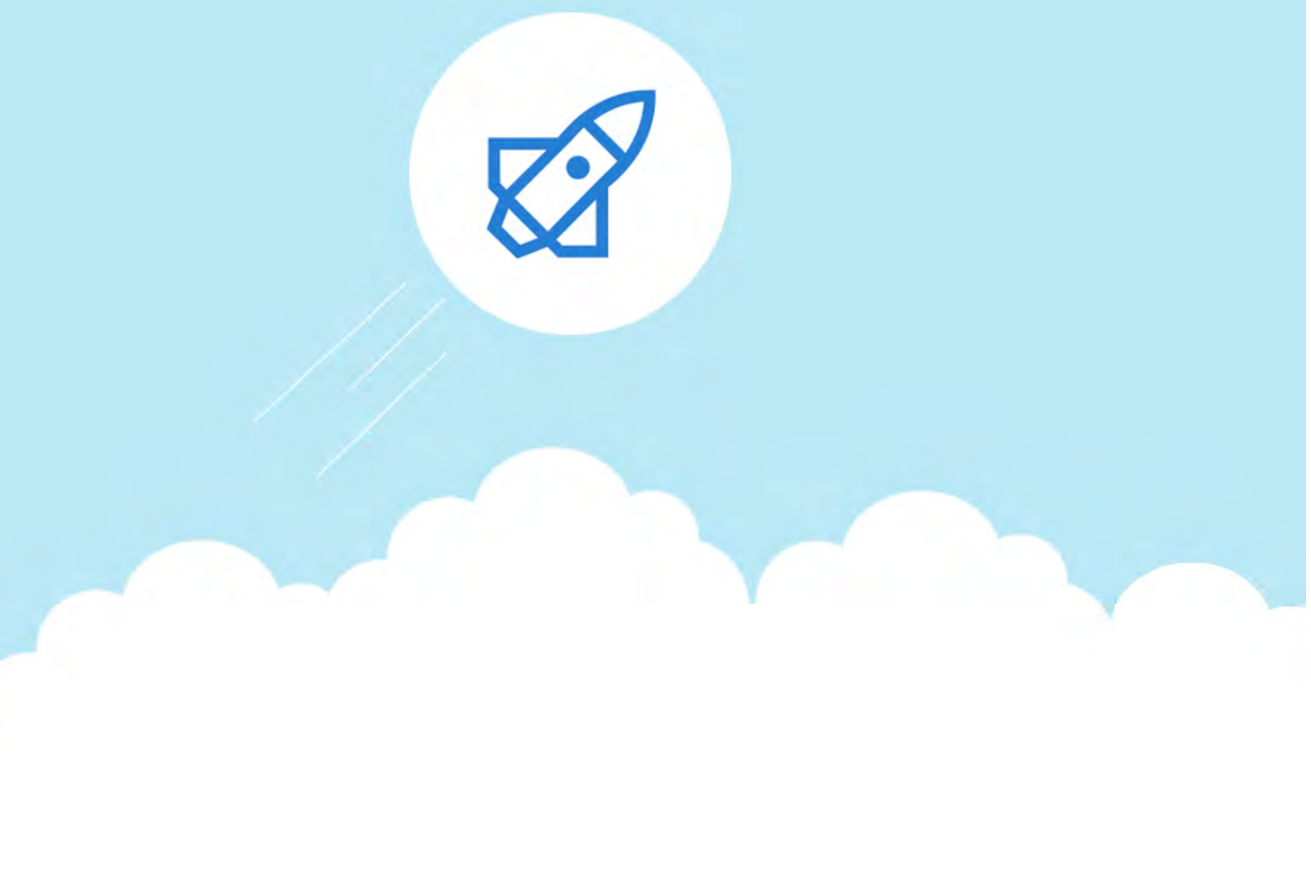
Yacoub Massad is a software developer who works mainly with Microsoft technologies. Currently, he works at Zeva International where he uses C#, .NET, and other technologies to create eDiscovery solutions. He is interested in learning and writing about software design principles that aim at creating maintainable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

Thanks to Damir Arh for reviewing this article.

Sanjeev Assudani

AZURE LOGIC APPS

An Overview



Introduction

In any enterprise, workflows and processes are important to run a business. Enterprises need a mechanism for automating the different business processes and integrating multiple applications they have. With the increase in cloud adoption, there is a growing need for integrating applications, data and workflows in cloud.

Microsoft Azure Logic App Services provides a mechanism for doing such integrations and business process automations, thereby creating workflows by orchestrating Software as a Service (SaaS) components.

In this article, I will provide an Overview, Key Components, Advantages and a Practical Scenario for Microsoft Azure LogicApps.

What are Azure LogicApps

The Azure Logic Apps provide a mechanism for application integration and workflow definition in the cloud. It provides a visual designer for configuring the workflows. You can define a workflow with connectors and logic creation using inbuilt standard connectors and enterprise integration connectors.

Logic Apps is a fully managed IPAAS (Integration platform as a service) with inbuilt scalability. You do not have to worry about hosting, scalability, availability and management. Logic Aps will scale up automatically to manage demand.

It is a Serverless application.

By serverless it doesn't mean there are no servers, it just means the developers do not have to worry about the underlying infrastructure. They just have to focus on the business logic. This results in a faster development and code simplicity.

At the sametime, it comes up with a consumption-based pricing plan, which means that the application is not charged if is it never used. All actions that run in a LogicApp are metered.

Azure LogicApp comes up with a workflow definition Language to describe its schema. The structure of the schema is as follows:

```
{ “$schema”: “”,
“contentVersion”: “”,
“parameters”: { },
“triggers”: [ { } ],
“actions”: [ { } ],
“outputs”: { }
}
```

Logic Apps Pricing

In Logic Apps, all the actions and trigger that are part of the workflow, are metered.

Logic Apps supports a volume based tiered model. The price is computed based on the actions executed.

Here is the pricing sheet for the Central US region and currency as INR.

Region: Central US Currency: Indian Rupee (₹)

Logic Apps Actions

ACTIONS EXECUTED / MONTH	PRICE PER ACTION EXECUTION
First 250K actions	₹0.052877 / action
250K-1M actions	₹0.026439 / action
1M-50M actions	₹0.009915 / action
50M-100M actions	₹0.005949 / action
100M+ actions	₹0.00357 / action

The storage and networking cost would be charged separately.

Microsoft Azure LogicApps Components

- **Workflow** - is used to define the different steps of your business process, using a graphical user interface.

- **Managed Connectors** - are used to connect to different data sources and services. Azure LogicApps provide an in-built set of managed connectors that cover different areas like social media, file FTP and many more.

- **Triggers** - initiate a workflow, and create a new instance of the workflow. A trigger can be an arrival of a file on FTP site or receiving an email. There are different types of triggers as mentioned below -

    1. **Poll triggers:** These triggers poll your service at a specified frequency to check for new data. When new data is available, a new instance of your logicapp runs with the data as input.
    2. **Push triggers:** These triggers listen for data on an endpoint, or for an event to happen, then triggers a new instance of your logicapp.
    3. **Recurrence trigger:** This trigger instantiates an instance of your logicapp on a prescribed schedule.

- **Actions** - An action represents a step in the workflow. It can invoke an operation on your API.

- **Enterprise Integration Pack (EIP) Connectors** - The EIP connectors provide functionality like BizTalk. These connectors are used for complex enterprise integration scenarios such as XML messaging and Validation. It supports exchange messages through industry-standard protocols, including AS2, X12, and EDIFACT. It requires an "Integration Accounts" to store different artefacts, like schemas, partners, certificates, maps and agreements.

Azure LogicApps - Advantages

Azure LogicApps have different advantages. I have mentioned some of these that I personally feel are important to below -

- **Inbuilt Connectors** - There are plenty of inbuilt connectors, triggers and actions that cover many of the common business scenarios. An example is performing sentiment analysis for a topic on Twitter.

- **Minimal development efforts** - With the graphical designer available in a browser as well as in Visual Studio, you can define and configure complex business processes with minimal development or no development efforts at all. The LogicApp automatically generates the code in the background.

- **Extensibility** - If the list of connectors do not meet your need, you can create your own custom APIs, Azure Functions and integrate them to the workflow.

- **Integration** - You can connect disparate systems together. You can integrate your on-premise legacy systems with the new ones in the cloud. Also, you can integrate different LogicApps together.

- **Templates** - There are plenty of predefined templates that are available for common business cases, like HTTP Request-Response, that greatly simplify the work of the development team. Templates are the fastest way to get started with the power of LogicApps.

- **DevOps Support** - It supports the continuous integration and continuous deployment methodology.

- **Advanced Integration** - It supports mature integration scenarios with the power of a XML messaging, trading partner management and more. You can leverage the power of BizTalk, which is a leading industry integration solution.

- **Inbuilt Diagnosis** - For any issues related to LogicApps, the Azure portal provides many tools to diagnose each logicapp at each step. You can see the trigger history and drill into that to see why the trigger didn't instantiate a LogicApp.

Logic Apps - Practical Scenario

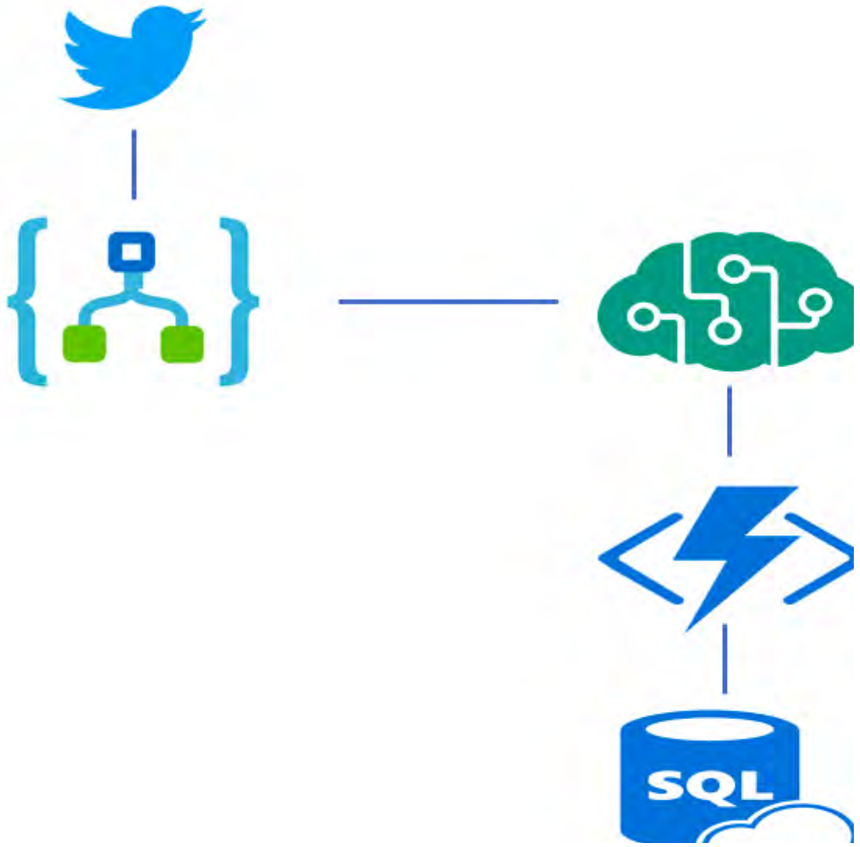
I will demonstrate the capabilities of Azure LogicApps using a practical scenario. In any organization, the chief marketing officer (CMO) would like to understand the sentiments for their different products and services on any social media medium, for example Twitter. To achieve this requirement, we can use Azure LogicApps and other Microsoft Azure services, as mentioned below.

Goal

Our goal is to perform sentiment analysis for a hashtag on Twitter (say #Azure) and store the results into SQL database.

Proposed Solution

To implement this, we will develop an Azure Logic App that will use Azure Cognitive Services to perform sentiment analysis. We will also use Azure Functions to do some computation. Here is the schematic view of this scenario.



Solution Implementation

For implementing this scenario, you need a Microsoft Azure subscription. You can implement it through a trial subscription as well.

In a browser window, open Microsoft Azure portal – https://portal.azure.com. Login using your credentials. Perform the steps mentioned below.

Step 1 – Create a new cognitive services of type Text Analytics (preview).
New -> Data + Analytics -> Cognitive Services.

This service will be needed as a step in the Logic App that we will create.

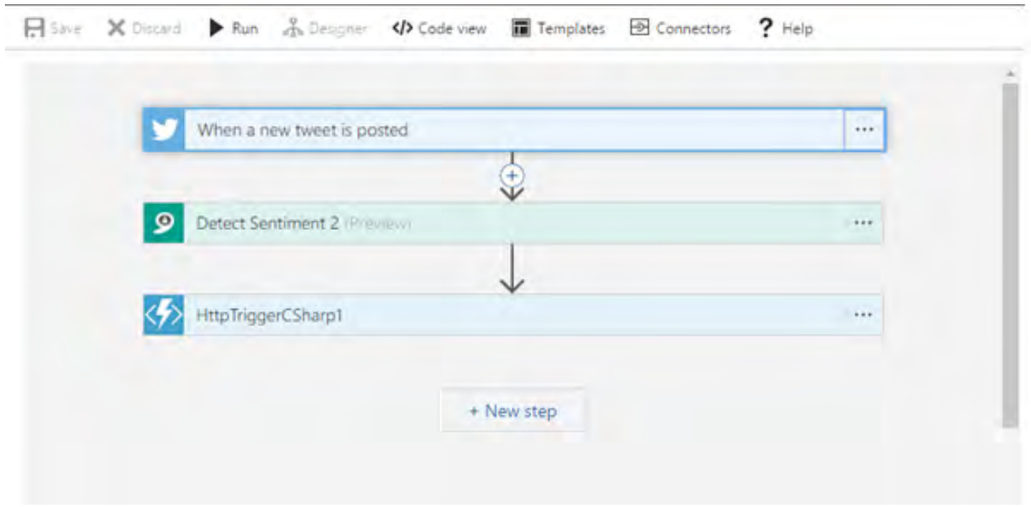
Step 2 – Create a new SQL database server.
New -> Database -> SQL Database.
We will need this to store the sentiment score and category.

Step 3 – Create a new Azure function by selecting the following
New -> Compute -> Function App.

This function will compare the sentiment score returned by the cognitive service we created in Step 1 and give it a category – "Red" or "Green". In this function we will also save the sentiment category to SQL database we created in Step 2.

Step 4 – Create a new Azure Logic App by selecting the following
New -> Web + Mobile -> Logic App.

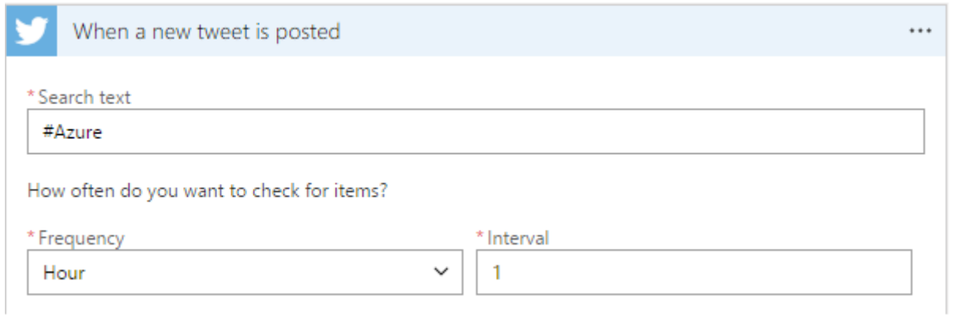
We will need this Logic App to glue the different services that we created in the steps above. Once the new Logic App is created, click on the Logic App Designer to add the different steps.



Mentioned here are the details for different steps in the workflow.

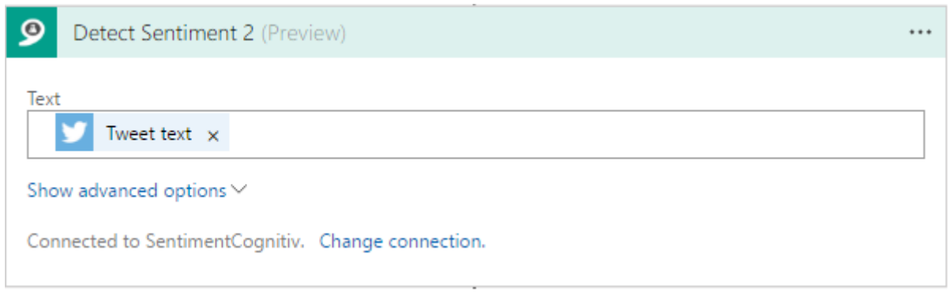
Step 1 – Twitter Trigger

This step will listen for the hashtag #Azure every hour. Once a tweet is posted, it will be passed to Step 2.



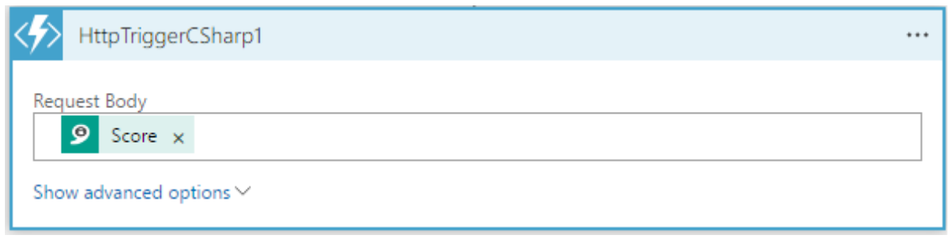
Step 2 – Azure Cognitive Services

Identify the sentiment associated with the twitter text. This will calculate the score and pass it to Step 3 - Azure Function.



Step 3 – Azure Function

Receive the score from the Step 2 and classify the score as Red or Green. For simplicity, I have classified the score of greater than 0.3 as Green. This function will also store score and classification details to SQL database.



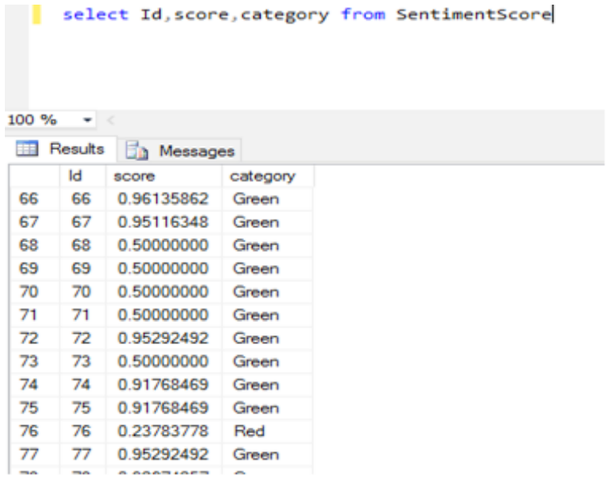
Here is the code from the Azure function –

```
#r "System.Data"
#r "System.Configuration"
using System.Net;
using System.Data.SqlClient;
using System.Configuration;

public static async Task<HttpResponseMessage> Run(HttpRequestMessage req,
TraceWriter log)
{
  log.Info("C# HTTP trigger function processed a request.");
  string category = "Red";
  dynamic data = await req.Content.ReadAsAsync<object>();
  log.Info(data.ToString());
  double sentimentScore;
  double.TryParse(data.ToString(),out sentimentScore);
  log.Info(sentimentScore.ToString());
  if (sentimentScore > 0.3)
    category = "Green";
    log.Info(category);

    var connString  = ConfigurationManager.ConnectionStrings["sqldb_connection"].
    ConnectionString;
    using(SqlConnection logicAppsConnection = new SqlConnection(connString))
    {
      string insertCommand = "Insert into SentimentScore (score,category) values (@
      score,@category)";
      logicAppsConnection.Open();
      using (SqlCommand command = new SqlCommand(insertCommand,
      logicAppsConnection))
      {
        command.Parameters.Add("@score", sentimentScore);
        command.Parameters.Add("@category", category);
        command.ExecuteNonQuery();
      }
      log.Info("Score details added to database successfully!");
    }
  return req.CreateResponse(HttpStatusCode.OK,category);
}
```

Some of the sample records from the database are shown below -



```
select Id,score,category from SentimentScore
```

100 %

Results | Messages

	Id	score	category
66	66	0.96135862	Green
67	67	0.95116348	Green
68	68	0.50000000	Green
69	69	0.50000000	Green
70	70	0.50000000	Green
71	71	0.50000000	Green
72	72	0.95292492	Green
73	73	0.50000000	Green
74	74	0.91768469	Green
75	75	0.91768469	Green
76	76	0.23783778	Red
77	77	0.95292492	Green

..and this completes a practical scenario implementation of Azure Logic Apps.
Just a word of caution, if you are not using your Azure Function, please disable it. If you keep it enabled and if it is executing, you will be charged.

Conclusion

Azure Logic Apps is a powerful, extensible and user friendly platform for developing enterprise integration applications. In my opinion, it can be used extensively for scenarios we just saw

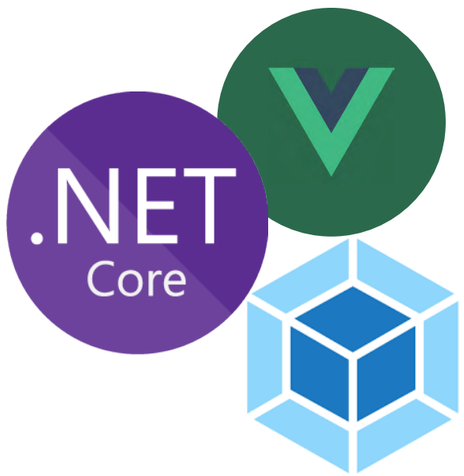
Sanjeev Assudani

Author

Sanjeev Assudani, is a Cloud Evangelist and a Microsoft Technology professional. He is currently working as a Technical Head at Calibre Mindware Programming Ltd, Pune, where he provides solutions to its customers on Microsoft .Net, ASP.Net MVC and Azure technologies. Sanjeev holds a Masters in Computer Sciences and Engineering from Oakland University, Rochester, MI, USA.
He can be reached at sanjeev.assudani@gmail.com.

Thanks to Vikram Pendse for reviewing this article.

Daniel Jimenez Garcia



MODERN WEB DEVELOPMENT USING ASP.NET Core TEMPLATE VUE.JS AND WEBPACK

If you read the DNC magazine regularly, you will probably be aware that we have covered .NET Core (bit.ly/dnc-dotnetcore), ASP.NET Core (bit.ly/dnc-aspnetcore) and have given an idea of what's upcoming (bit.ly/dnc-newdotnet) for .NET web developers.

A lot has been happening lately in the .NET Ecosystem!

For example, consider the JavaScript landscape of frameworks to help you organize your client-side code. As of 2017, there is a lot of fragmentation and different options (bit.ly/dnc-jsfwformvc) for .NET web developers out there!

One option increasing in popularity is Vue.js, which has already received some coverage (bit.ly/dnc-vuejs) in this magazine. Vue is a lightweight and performant framework for creating user interfaces that was originally created and open sourced by Evan You, after working for Google on Angular.

Image Courtsey: Wikimedia Commons

This article explores the official Microsoft template for ASP.NET Core that uses Vue as its client-side framework, which is also a part of their JavaScriptServices project (bit.ly/dnc-jsservices). It should get you started on the right track without manually configuring modern tooling and libraries like Webpack, Babel or hot-reload.

Even if you are not particularly interested in Vue.js, I think you will find plenty of useful information on topics like Webpack and debugging that would apply to other project templates for Angular and React.

You can download the article code from GitHub (bit.ly/dncm31-aspcorevue).

Why Vue?

You might be wondering why do you need to worry about yet another JavaScript framework when you already know the likes of Angular or React. And the answer to that question would be that if you are already using Angular or React and you are happy with them, then you p*robably don't need to* worry about Vue.

However, you might be now evaluating different options for your next project, or the next de-facto stack for your company. Or maybe you are not entirely happy about your current framework, and you think something lighter/faster/simpler could be worth considering.

Of course, you might just be curious to learn what's the fuss about this framework!

Editorial Note: *If you want a detailed comparison of Vue vs Angular vs React, check out VueJS vs Angular vs ReactJS with Demos (bit.ly/dnc-vueangreact)*

In Vue, you will find a very lightweight and performant framework focused at its core on the view layer. It is designed to be fast and optimized by default, using a light-weight virtual DOM implementation. According to the official comparison with other frameworks, you could think about Vue having React performance with a `shouldComponentUpdate` function automatically implemented for you on every component.

- Apart from being fast, its declarative syntax using plain JavaScript objects makes building reactive user interfaces a breeze.

- It comes with a very well-designed components module. Decomposing your UX into a tree of components

is very straightforward.

- You decide which style works better when it comes to mapping your components to files: Single .**vue** file? Separated files for the template, code and style rules? Inline definition of components?

Let's check an example from the official Vue guide (try it on jsFiddle here):

HTML code:

```
<div id="app">
 <ol>
  <todo-item
   v-for="item in todoList"
   v-bind:todo="item"
   v-bind:key="item.id">
  </todo-item>
 </ol>
</div>
```

JS code:

```
Vue.component('todo-item', {
  props: ['todo'],
  template: '<li>{{ todo.text }}</li>'
});
var app = new Vue({
  el: '#app',
  data: {
    todoList: [
      { id: 0, text: 'Learn Vue' },
      { id: 1, text: 'Download ASP.Net Core 2.0' },
      { id: 2, text: 'Join the Node.js meetup' }
    ]
  }
});
```

The best part is that Vue doesn't really end in the view layer.

> *Vue is designed so you can adopt its core library and use it for building your view layer, but at the same time, it can be extended with plugins and components that let you build complex SPA experiences.*

This puts you in control of what you need to exactly bring into your project, with absolute liberty, to pick and match depending on your needs.

A curated list of components and libraries is provided in the awesome-vue GitHub page, but of course you can use other (or your own) libraries.

Before we move on, let me point you again towards this comparison section of the official Vue documentation, as well as this comparison article on DotNetCurry by Benjamin. If you are still confused about how Vue might help you in your projects, use these articles to compare it against the most popular JavaScript frameworks.

Installing the SPA templates

The first thing we need to do is to install the SPA (Single Page Application) templates provided by Microsoft. Following their initial blog post, this is as easy as running:

```
dotnet new --install Microsoft.AspNetCore.SpaTemplates::*
```

Now if you run `dotnet new` in a console, you will see a few different SPA templates. In its current version, this package installs templates for Vue, Angular, React, Knockout and Aurelia:

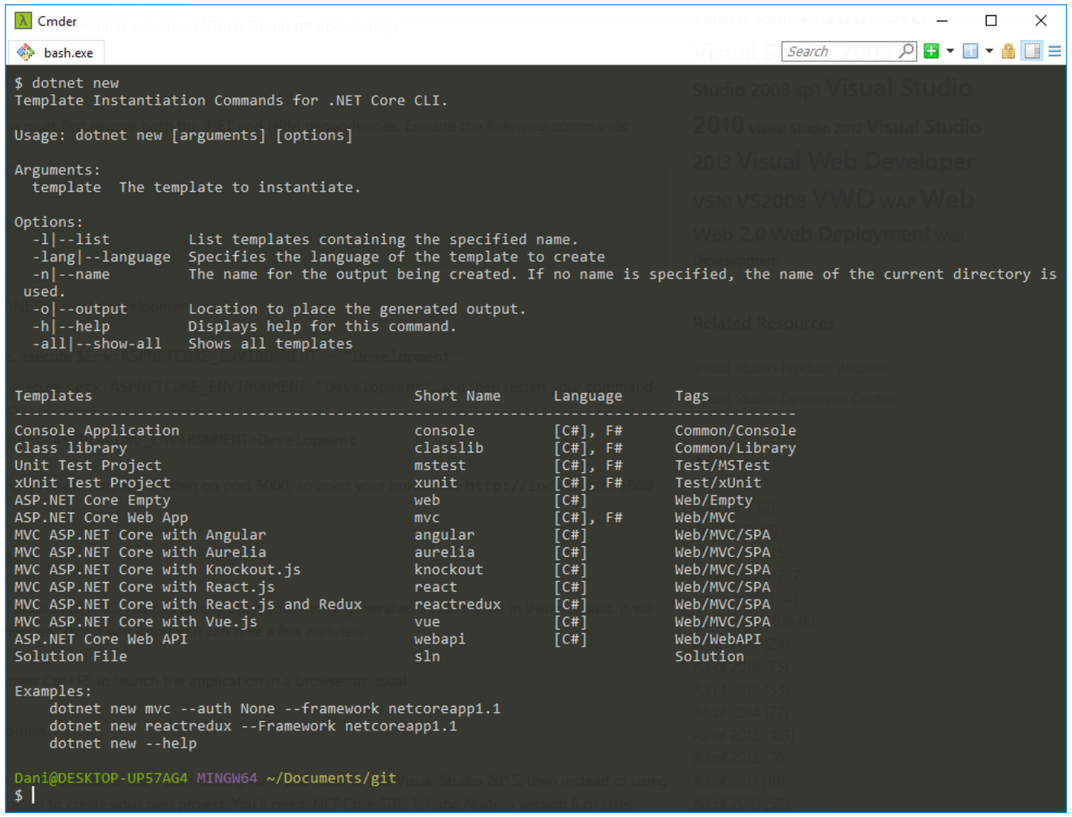


Figure 1, installed "dotnet new" templates

Once installed, you are ready to go, simply run `dotnet new <template-name>` on the console

Creating a new ASP.NET Core project with Vue

As we saw in the previous section, the official Microsoft SPA templates provide a template with Vue as the client-side framework.

To get started and create a new Vue project, simply run the following commands on the console:

```
mkdir new-project
cd new-project
dotnet new vue
```

That will generate a new ASP.NET Core project using the Vue SPA template. Now let's run the project and verify everything is working.

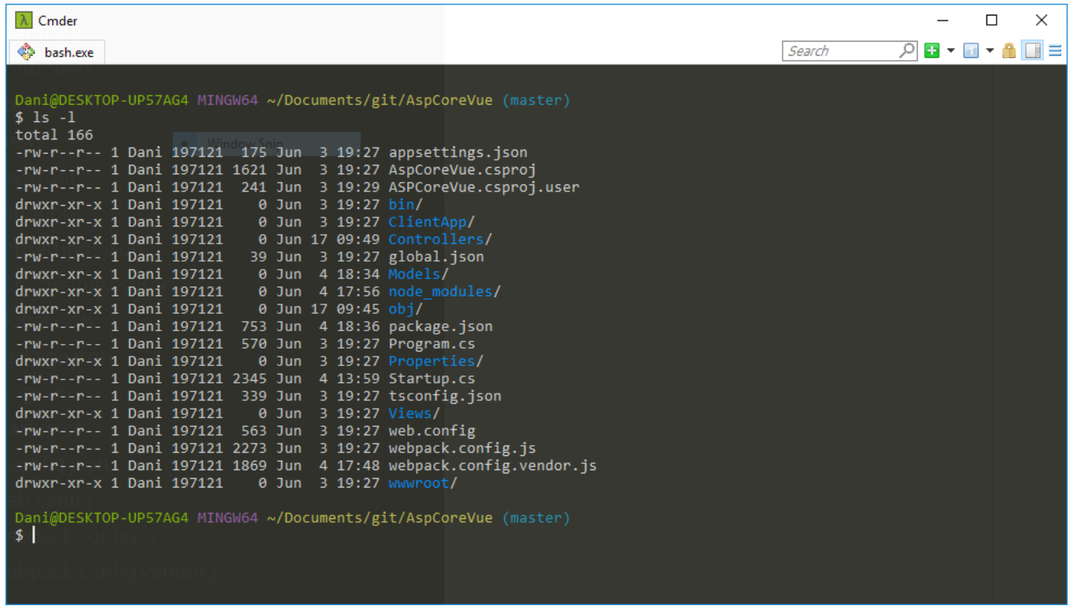


Figure 2, the generated project

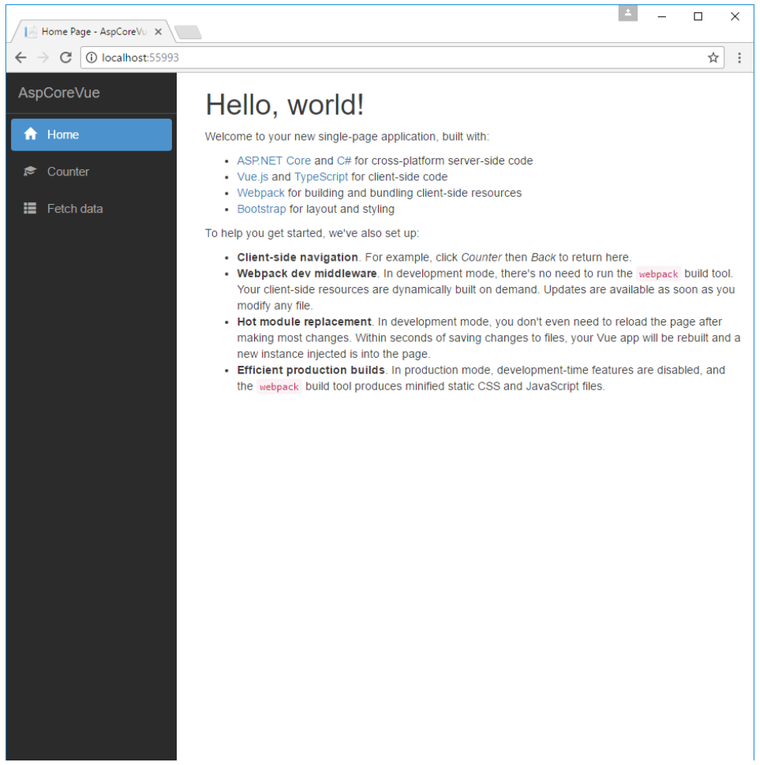


Figure 3, running the project

Running the project from Visual Studio 2017

Everything has been wired in the project template, so locally running your new application works Out of the Box as you would expect.

Open the project with Visual Studio 2017 and press F5.

- NuGet and npm dependencies will be restored/installed if it's the first time you opened and launched the project
- The application will be built with the Debug configuration
- Once ready, your browser will be launched and your application will be loaded.

The debugger is attached as you would expect, and that includes setting breakpoints both in the server-side code and the client-side TypeScript Vue code!

Note: *Client-side debugging with VS would only work on supported browsers like Chrome and Edge!*

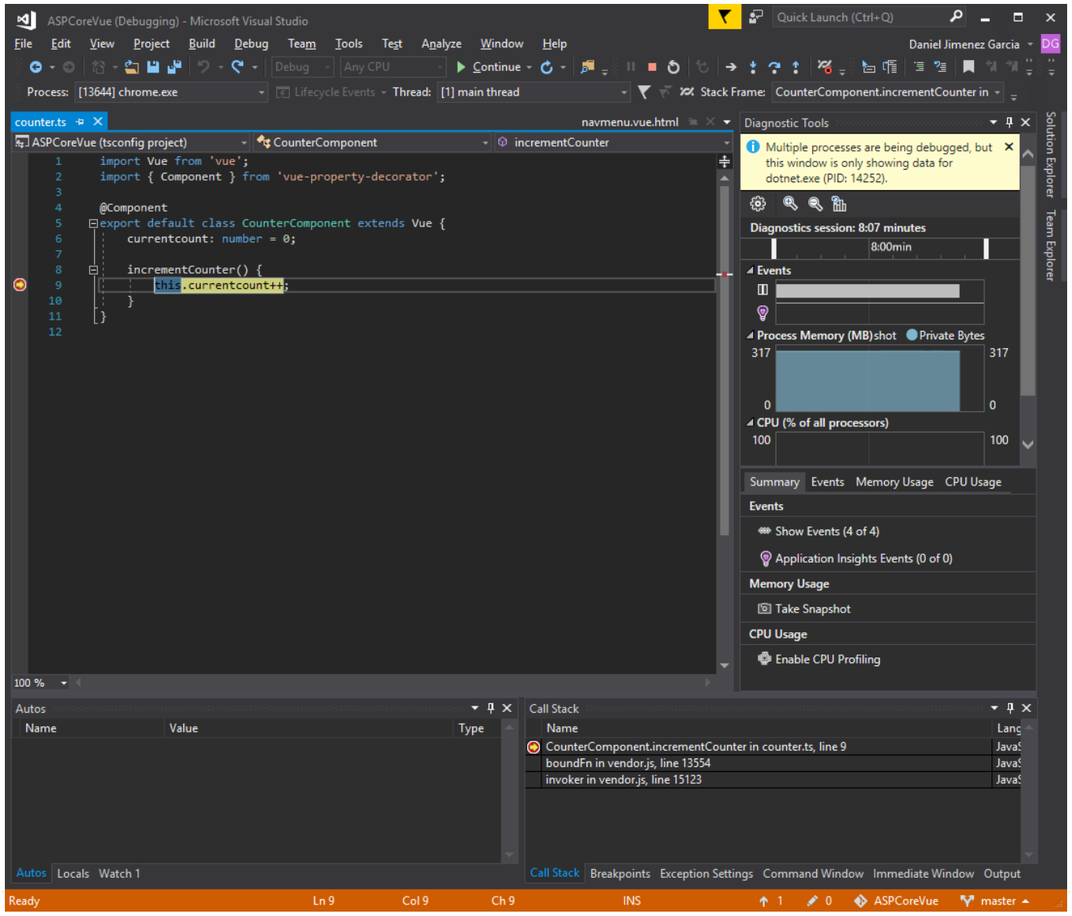


Figure 4, breakpoint on a TypeScript file in Visual Studio 2017

Running the project from Visual Studio Code

Visual Studio Code (VS Code) is one of the better products Microsoft has come up in recent times and it is no surprise it is being increasingly adopted by many developers. I have personally been using it for creating several web applications over the last year and I am enjoying every aspect of it.

How well does VS Code play with our newly created Vue project?

You need to implement a few steps before you get the same debugging experience of VS 2017, in VS Code:

- Make sure you have the **C#** and **Debugger for Chrome extensions** (bit.ly/dnc-m31-vsdebug)

- **Run `npm install` on your project root to restore all the required node modules, like webpack and its dependencies**

- Follow the steps on this related article on Debugging ASP.NET Core SPA in Visual Studio Code to create the necessary launch.json and tasks.json configuration files.

If you have any trouble following that article and creating these files (or don't have the time for it), check the **.vscode** folder in the companion code in GitHub.

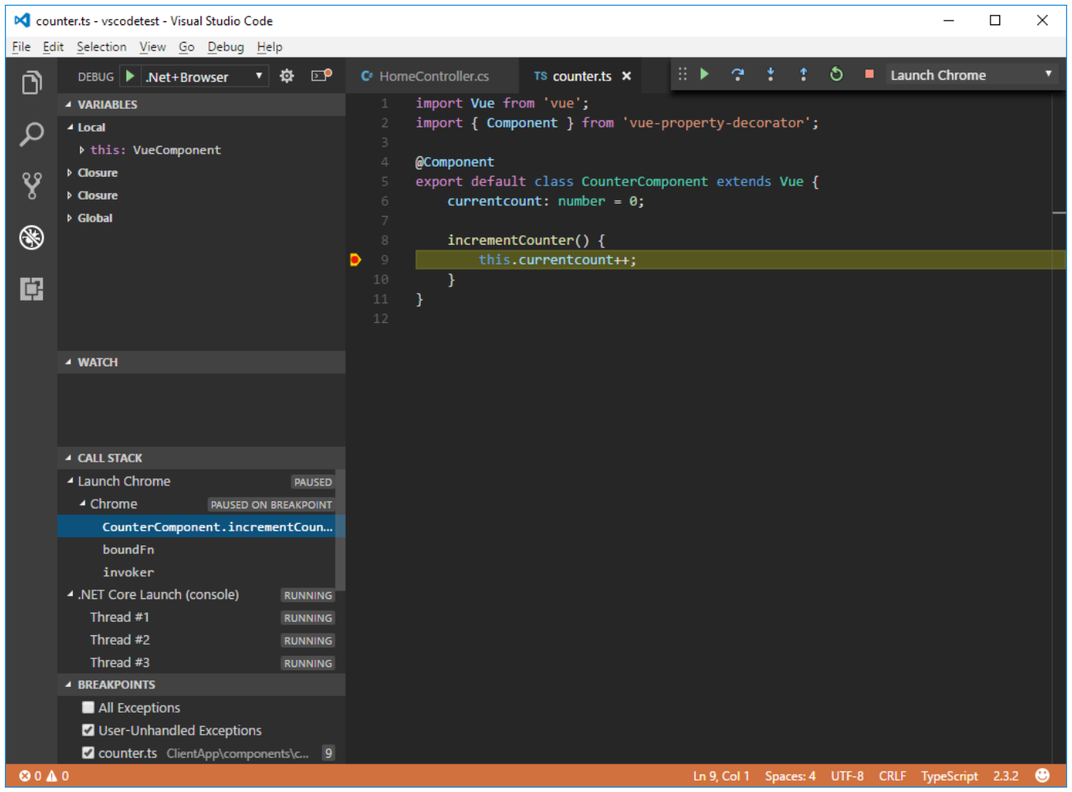


Figure 5, debugging client and server side in VS Code

Running the project from the Command Line

Another valid option to start your new and shiny project without special debugging support is by running it directly on the command line.

- You need to run `dotnet restore && npm install` after the project was created. This installs the required NuGet and npm dependencies.

- To properly run in debug mode, you need to set the environment variable ASPNETCORE_ENVIRONMENT to "Development". See the announcement blogpost for more details.

Then execute `dotnet run` on the command line and open http://localhost:500 in your browser. Stop with Ctrl+C once you are done.

Out of the Box features

In this section, we will take a closer look at some of the features, libraries and tooling included by default within the Vue template.

At a high level, what you have is a project made of:

- ASP.Net Core backend
- SPA frontend using Vue 2 and TypeScript
- Bootstrap 3 styling
- webpack generating the js/css bundles ultimately send to the browser, wired for both development and production modes.

Let's start somewhere familiar and check the Index method of **Controllers/HomeController.cs**. All this does is rendering the Index view. Now let's check the Index view and you will see it does very little:

- Renders the html element (see that div with id **app-root**?) where our Vue app will be mounted
- Loads the Webpack bundle that contains the transpiled JavaScript code of our Vue application

```
@{
  ViewData["Title"] = "Home Page";
}

<div id='app-root'>Loading...</div>

@section scripts {
  <script src="~/dist/main.js" asp-append-version="true"></script>
}
```

If you are unfamiliar with webpack, we will take a closer look at it later; for now, see it as the piece of the project that takes all the TypeScript/JavaScript and CSS/LESS files of the client-side code and produces the combined bundle files that are sent to the browser.

Ok, so now let's open **ClientApp/boot.ts**. This is the entry point for our client side code, the file that contains the code starting the Vue application in the browser:

```
import 'bootstrap';
import Vue from 'vue';
import VueRouter from 'vue-router';
Vue.use(VueRouter);

const routes = [
  { path: '/', component: require('./components/home/home.vue.html') },
  { path: '/counter', component: require('./components/counter/counter.vue.html')},
  { path: '/fetchdata', component: require('./components/fetchdata/fetchdata.vue.
html') }
];

new Vue({
  el: '#app-root',
  router: new VueRouter({ mode: 'history', routes: routes }),
  render: h => h(require('./components/app/app.vue.html'))
});
```

As you can see, it is loading the main 3rd party libraries we depend on (mainly Bootstrap and Vue) and starting the Vue app attached to the html element **#app-root**. Remember, this element was rendered by the **Home/Index.cshtml** view.

It is also establishing the client-side routes by instantiating its own **VueRouter**. This means when you navigate to one of those pages like http://localhost:5000/counter, the vue specific router will take care of instantiating and rendering the vue component found in **ClientApp/components/counter/**.

And where will it be rendered?

Well, if you take another look at the template inside **app.vue.html**, which is the template for the main app component instantiated in **boot.ts**, you will notice it has a placeholder for `<router-view></router-view>`:

```
<template>
  <div id='app-root' class="container-fluid">
    <div class="row">
      <div class="col-sm-3">
        <menu-component />
      </div>
      <div class="col-sm-9">
        <router-view></router-view>
      </div>
    </div>
  </div>
</template>
```

Wait a second!! This means we have client-side routes and server-side routes?

Yes, that is correct!

But what if I navigate directly to http://localhost:5000/counter, won't that be handled by the ASP.NET Core routing and return a 404?

This is taken care by how the routes are being wired on the server side. Open **Startup.cs** again and take a look at the routes being registered.

You will notice a **spa-fallback** route being added at the end:

```
app.UseStaticFiles();
app.UseMvc(routes =>
{
  routes.MapRoute(
    name: "default",
      template: "{controller=Home}/{action=Index}/{id?}");
  routes.MapSpaFallbackRoute(
    name: "spa-fallback",
      defaults: new { controller = "Home", action = "Index" });
});
```

This routing helper is part of Microsoft's JavaScript services, more concretely of their Microsoft.AspNetCore.SpaServices package. So when a request like localhost:5000/counter arrives, the following happens:

1. The static files middleware is executed and does nothing since it doesn't match any static file.

2. The routing middleware is executed next, and tries the first default route. Since the url doesn't match to any known controller-action pair, it does nothing.

3. The next route is executed, the spa-fallback one. And this will basically execute Home's Index action, which in turn renders the **#app-root** element and our client-side app code. This way when the browser executes our vue startup code part of **boot.ts**, our *VueRouter* will react to the current url by rendering the counter component!

The only exception made by the spa-fallback route is when the url appears to be something like http://localhost:5000/missing-file.png.

In this case, it will do nothing and the request will end in 404 which is probably what you want. This might be a problem if your client-side routes contain dots, in which case the recommendation is that you add a catchall MVC route rendering the Index view. (But bear in mind in this case requests for missing files will actually render your index view and start your client side app).

Webpack

Webpack is quite an important piece in all this setup. It is the bridge between the source code of your client-side app located in **/ClientApp**, and the JavaScript code executed by the browser.

If you are new to webpack, I think by now you will have at least two questions if not more:

- The source code of the client side app is written in Typescript. How/where is this converted to JavaScript before being sent to the browser?

- The source code of the client app is structured in multiple files. Furthermore, most components have separate template and TypeScript files. However, in our razor views, we only render a single script with `src= "~/dist/main.js"`. How is this file created?

This is all webpack's work behind the scenes. Webpack is defined as a *module bundler* meaning it understands your source code composed of many individual files (TypeScript, JavaScript, less, sass, css, fonts, images, and more!) and produces a combined file ready to be used in the browser.

How this happens, is defined in a couple of files on your project root folder named **webpack.config.js** and **webpack.config.vendor.js**.

Let's start by inspecting webpack.config.js.

Exploring webpack.config.js

See the **entry** property of the configuration object? That's the initial file you point webpack to for each of the bundles to be generated, and it will then find your other files through the import statements on them. We are basically configuring a single bundle named "main", to be composed of files starting from ClientApp/boot.ts:

```
entry: { 'main': './ClientApp/boot.ts' },
```

Now jump to the **output** property.

Remember that in the razor Index view there was a single script statement with source equals to **~/dist/main.js**? That was because we are telling webpack to generate the bundles inside the **wwwroot/dist** folder of the project and use the name of the bundle as the file name:

```
output: {
  path: path.join(__dirname, bundleOutputDir),
  filename: '[name].js',
  publicPath: '/dist/'
},
```

So the *entry* property defines the bundles that will be generated (in this case a single one named main) and the output property defines where these bundles will be physically generated.

The next stop is the **module** property of the configuration. By default, webpack only understands JavaScript code.

However, our source code is made of TypeScript files, html templates, css files, etc. We need to tell webpack how to interpret these files so they can be processed and included in the generated bundle!

We do so by configuring rules that match certain file types with specific loaders.

```
module: {
  rules: [
    { test: /\.vue\.html$/, include: /ClientApp/,
      loader: 'vue-loader',
      options: {
        loaders: { js: 'awesome-typescript-loader?silent=true' } } },
    { test: /\.ts$/, include: /ClientApp/,
      use: 'awesome-typescript-loader?silent=true' },
    { test: /\.css$/, use: isDevBuild ?
      [ 'style-loader', 'css-loader' ] :
      ExtractTextPlugin.extract({ use: 'css-loader?minimize' }) },
    { test: /\.(png|jpg|jpeg|gif|svg)$/, use: 'url-loader?limit=25000' }
  ]
},
```

These loaders know how to treat these files. For example:

- The **vue-loader** understands files divided into <template>, <script> and <style> sections and interprets each of these parts

- The **awesome-typescript-loader** understands TypeScript files and will transpile them into JavaScript using the options in the tsconfig.json file of your project.

Finally, we have the plugins section where we can configure webpack for more complex work. For example:

- We can let webpack know about a separated "vendor" bundle that will contain libraries code like that of Vue or Bootstrap, to keep our main bundle containing only our app specific code.

- We can add source maps when generating the bundles for development, to debug the generated bundles in the browser

- We can optimize the bundles for production further minifying and uglifying the produced bundles.

By now you will probably be curious about what the separated **webpack.config.vendor.js** file is for. This is a similar file that takes care of processing 3rd party libraries like Vue or Bootstrap, generating a couple of files vendor.js and vendor.css files at the end.

If you take a look at the razor **_Layout.cshtml** file, you will notice it is rendering a stylesheet with `href="~/dist/vendor.css"` and a script with `src="~/dist/vendor.js"`. These files are generated there when webpack is run using the **webpack.config.vendor.js** settings.

There is nothing new on that file, except maybe that CSS code is extracted into its own file (vendor.css) rather than being inlined in the same file (like it happens for the main bundle).

Note: This setup assumes you will rely on bootstrap default styles with a few small custom rules on your specific code, but if you had large style definitions, you might consider modifying webpack.config.js to also extract css rules into its own main.css file.

There is a lot more to webpack and many other options/scenarios available than what I have just covered here. Hopefully it will be enough to give you a rough idea of how things work.

If all of this seems very confusing, don't worry; you are not the only one feeling that way. It will make sense as you use it more and come across more scenarios where webpack can help you.

Webpack in development

We have been running the project earlier and we didn't have to generate those webpack bundles. In fact, we didn't have to worry about webpack at all.

This is because of a very helpful middleware part of Microsoft.AspNetCore.SpaServices, which is wired by default into the Vue template (as well as other SPA templates)!

I am referring to the Webpack dev middleware, which you can see being wired in the **Startup.cs** file:

```
if (env.IsDevelopment())
{
  app.UseDeveloperExceptionPage();
  app.UseWebpackDevMiddleware(new WebpackDevMiddlewareOptions {
    HotModuleReplacement = true
  });
}
```

What this does, is to intercept requests for files in **~/dist/** and automatically generate the bundles. Even more interesting is the **HotModuleReplacement** option. With this option enabled, the webpack dev middleware will monitor changes to the TypeScript files, templates, css etc. of the client-side app. When any file changes, the bundles will be automatically regenerated and pushed to the browser which will reload only the code that changed!

For further reading, check the documentation on Microsoft's middleware repo. You will find out that .NET middleware mostly delegates to webpack's hot module replacement middleware which is used under the hood.

Webpack in production

When publishing your app to production, the aim is to produce bundles which are as optimized as possible. This means developer tools like source maps won't be used in the production bundles, but it also means additional plugins can be used to minify/compress the generated bundles. We have already seen these while going through the webpack config files.

So how is this production specific build invoked? The project comes with a publish task already wired that will basically run webpack in production mode and generate production optimized bundles in the **wwwroot/dist** folder.

You can see this if you open the **.csproj** file and look at the **RunWebpack** task:

```
<Target Name="RunWebpack" AfterTargets="ComputeFilesToPublish">
  <!-- As part of publishing, ensure the JS resources are freshly built in
production mode -->
  <Exec Command="npm install" />
  <Exec Command="node node_modules/webpack/bin/webpack.js --config webpack.config.
vendor.js --env.prod" />
  <Exec Command="node node_modules/webpack/bin/webpack.js --env.prod" />

  <!-- Include the newly-built files in the publish output -->
  <!-- Removed for clarity -->
</Target>
```

So, if you manually run `dotnet publish -c Release` on the command line or use the Publish menu of Visual Studio, webpack will be used to generate production optimized bundles and these will be included within the published files of your app.

Show me some code!

The template's fetchdata page

Let's start by taking a look at the component located in **ClientApp/components/fetchdata**. This is the Vue component that gets rendered when you hit the /fetchdata route as in http://localhost:5000/fetchdata. The interesting part is that this component will automatically fetch some JSON data from the ASP.NET Core backend as soon as it gets rendered.

It does so by providing a function named `mounted` which sends an ajax request to fetch the data, and then updates the component's data. The mounted function is one of the many lifecycle hooks provided by Vue and you don't need to do anything special other than making sure your component has a function with the required name.

```
@Component
export default class FetchDataComponent extends Vue {
    forecasts: WeatherForecast[] = [];

    mounted() {
        fetch('/api/SampleData/WeatherForecasts')
            .then(response => response.json() as Promise<WeatherForecast[]>)
            .then(data => {
                this.forecasts = data;
            });
    }
}
```

The fetch method is provided by the isomorphic-fetch library which is part of the vendor's webpack bundle and adds that fetch function to the global scope.

All the component's template needs to do is render each item of the forecast's array, for example in a table:

```
<tr v-for="item in forecasts">
    <td>{{ item.dateFormatted }}</td>
    <td>{{ item.temperatureC }}</td>
    <td>{{ item.temperatureF }}</td>
    <td>{{ item.summary }}</td>
</tr>
```

Given Vue's reactive nature, as soon as the forecast array is updated in the component, the template is re-rendered. This way the items appear on the browser as soon as they were fetched from the server.

Hopefully you agree with me, this looks straightforward. Continue reading if you wish to create something of our own based on what we have discussed so far.

Adding a TODO list

Let's implement the classical example of the TODO list, where we will add a new client-side route and components, backed by a REST API on the server side.

The API is the most straightforward and boring piece!

This is all very familiar to anyone who has created APIs either in ASP.NET Core or WebAPI. I have used in-memory storage to quickly have something up and running, but you could easily add a proper database storage.

```
[Route(“api/[controller]”)]
public class TodoController : Controller
{
  private static ConcurrentBag<Todo> todos = new ConcurrentBag<Todo> {
    new Todo { Id = Guid.NewGuid(), Description = “Learn Vue” }
  };

  [HttpGet()]
  public IEnumerable<Todo> GetTodos()
  {
    return todos.Where(t => !t.Done);
  }

  [HttpPost()]
  public Todo AddTodo([FromBody]Todo todo)
  {
    todo.Id = Guid.NewGuid();
    todo.Done = false;
    todos.Add(todo);
    return todo;
  }

  [HttpDelete(“{id}”)]
  public ActionResult CompleteTodo(Guid id)
  {
    var todo = todos.SingleOrDefault(t => t.Id == id);
    if (todo == null) return NotFound();

    todo.Done = true;
    return StatusCode(204);
  }
}
public class Todo
{
  public Guid Id { get; set; }
  public string Description { get; set; }
  public bool Done { get; set; }
}
```

With our API done, let’s add a new route and component to the client side.

Start by creating a new folder **ClientApp/components/todos**. Now let’s add a component file named **todos.ts**, with a new vue component:

```
import Vue from ‘vue’;
import { Component } from ‘vue-property-decorator’;

interface TodoItem {
  description: string;
  done: boolean;
  id: string;
}

@Component
export default class TodoComponent extends Vue {
  todos: TodoItem[];
  newItemDescription: string;

  data() {
    return {
      todos: [],
      newItemDescription: null
    };
  }
}
```

It doesn’t do much right now other than declaring a new component that will hold the list of todo items. Let’s also add a new template file named todos.vue.html with the contents:

```
<template>
  <div>
    <h1>Things I need to do</h1>
  </div>
</template>
<script src=”./todos.ts”></script>
```

Not very interesting right now, but let’s finish adding all the files and ensure we have everything wired correctly before adding the functionality.

Now add the new route in boot.ts adding the following entry to the routes array:

```
{ path: ‘/todos’, component: require(‘./components/todos/todos.vue.html’) }
```

Finally, open navmenu.vue.html and add another entry to the menu that renders the /todo route:

```
<li>
  <router-link to=”/todos”>
    <span class=”glyphicon glyphicon-list-alt”></span> TODO list
  </router-link>
</li>
```

Now run the project and you should see the entry in the menu rendering the todos component:

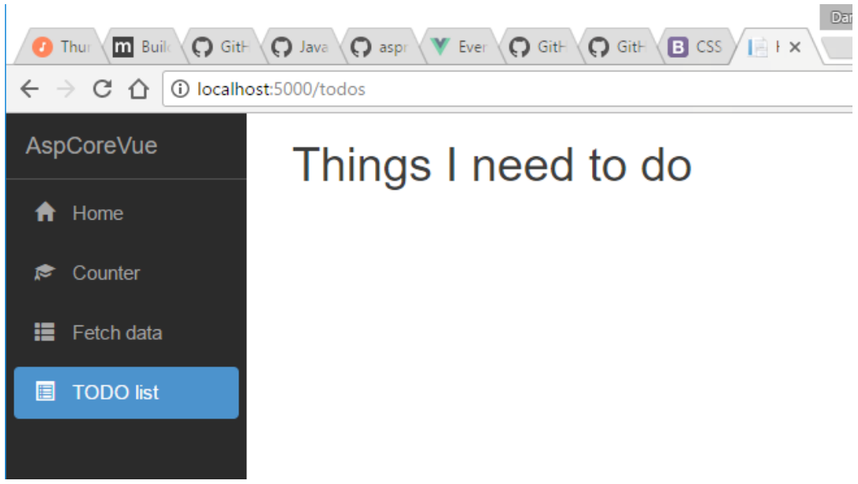


Figure 6, adding the todo route and component

Now that we have our todo component ready, let’s update it so it fetches the list of todos from the backend API and renders them in a list.

Simply add the **mounted** function to the component inside **todos.ts** and use the fetch method provided by isomorphic-fetch to load the todo items:

```
mounted() {
  fetch('/api/todo')
    .then(response => response.json() as Promise<TodoItem[]>)
    .then(data => {
      this.todos = data;
  });
}
```

Then use vue's **v-for** directive inside the template inside **todos.vue.html** to render each item as an <li> inside a list:

```
<ul v-if="todos.length">
  <li v-for="item in todos">{{item.description}}</li>
</ul>
<p v-else class="text-success">Nothing to do right now!</p>
```

Didn't work?

Double check if you have added that markup inside the outer `<div>` element which is the root of the template in **todos.vue.html**. Vue is very strict about having component's template with a single root element.

Ok, we can now render all the items!

Let's now add a button to complete each of these items. We can tweak the template, so for each item, we also add a button. This button will use vue's **v-on** directive to call a method of the component when the **click** event of the html button is triggered:

```
<li v-for="item in todos">
  {{item.description}} -
  <button v-on:click="completeItem(item)" class="btn btn-link">
    Completed
  </button>
</li>
```

The implementation of the completeItem method can be as simple as using the **fetch** method to send a DELETE request to the backend and remove the item from the list of todos:

```
completeItem(item: TodoItem){
  fetch(`/api/todo/${item.id}`, {
    method: 'delete',
    headers: new Headers({
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    })
  })
  .then(() => {
    this.todos = this.todos.filter((t) => t.id !== item.id);
  });
}
```

The remaining task would be to create a form to add a new todo item. Let's start by updating the template with a form:

```
<form>
  <div class="row">
    <div class="col-xs-6">
      <input v-model="newItemDescription" class="form-control"
      placeholder="I just remembered that I have to...">
    </div>
    <button v-on:click="addItem($event)" class="btn btn-primary">
      Add
    </button>
  </div>
</form>
```

The form has an input element whose value is bound to the `newItemDescription` property of the component's data. When the click event of the button is triggered, the `addItem` method of the component is called.

The method will send a POST to the backend API and update the list of todos with the new item:

```
addItem(event){
  if(event) event.preventDefault();

  fetch('/api/todo', {
    method: 'post',
    body: JSON.stringify(<TodoItem>{description: this.newItemDescription}),
    headers: new Headers({
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    })
  })
  .then(response => response.json() as Promise<TodoItem>)
  .then((newItem) => {
    this.todos.push(newItem);
    this.newItemDescription = null;
  });
}
```

After all these changes, you should have a SPA with a fully functional (albeit simple) todo page:

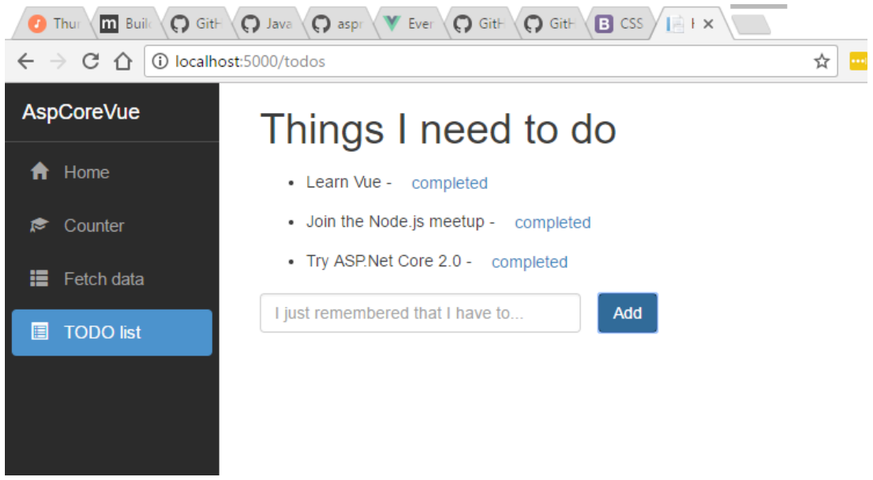


Figure 7, completed todo page

This isn't a comprehensive example, but it should be enough to get you started with Vue and the project template. After this, you should have an easier time exploring more advanced Vue functionality.

Conclusion

The modern JavaScript world is a complex and evolving one. It is hard to stay up-to-date and keep track of every new framework that appears and replaces something you just learned like grunt/gulp!

Thankfully, templates similar to the SPA ones provided by Microsoft, have already gone through the work of curating these tools and frameworks and have provide you with a very decent and up-to-date starting point. With these ASP.NET Core SPA templates, you can start adding value instead of spending a week understanding and configuring today's hot libraries.

Of course, at some point, you will need to understand these pieces, but you can do so gradually and it doesn't act as a barrier preventing you from starting a new project on the right track.

And Vue is a great option as a client-side framework for your next project. It strikes a lovely balance between simplicity, performance and power that makes it easy to get started, but gradually supports more advanced features and applications.

I understand choosing a framework is not an easy task and there are a lot of factors to consider like team skills or legacy code, but I personally would at least consider Vue even if I had to later discard it because of some valid team/company/project reasons.

I would love to hear about your experiences working with Vue and ASP.NET Core SPA templates ■

Download the entire source code from GitHub at
bit.ly/dncm31-aspcorevue

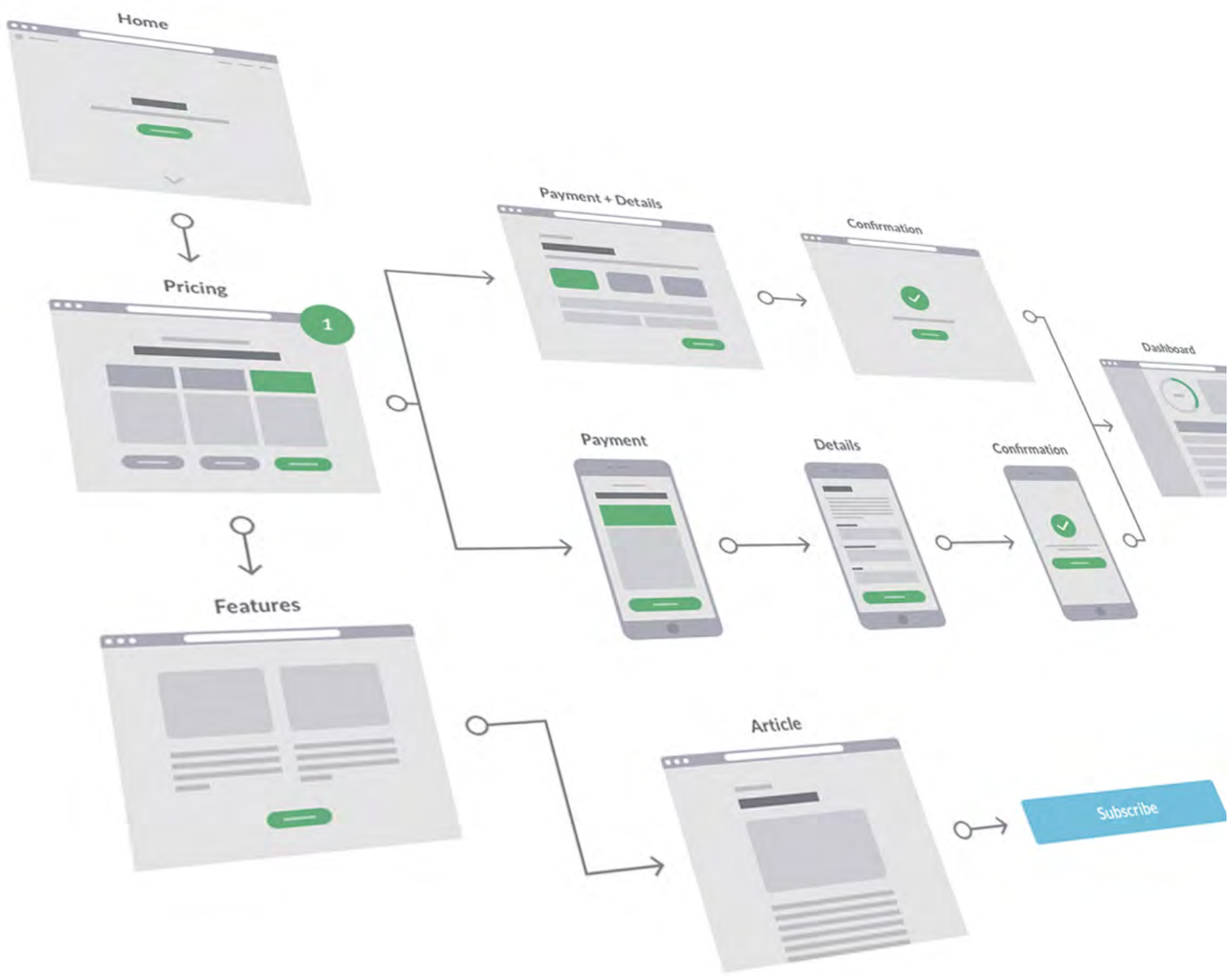
Daniel Jimenez Garcia
Author

Daniel Jimenez Garcia is a passionate software developer with 10+ years of experience. He started as a Microsoft developer and learned to love C# in general and ASP MVC in particular. In the latter half of his career he worked on a broader set of technologies and platforms while these days is particularly interested in .Net Core and Node.js. He is always looking for better practices and can be seen answering questions on Stack Overflow.

Thanks to Damir Arh for reviewing this article.

Rahul Sahasrabuddhe

UX HYPE OR REALITY?



Introduction

Consider these situations:

1) People who use iPhone are generally ardent fans of iOS and all other Apple products. They simply love the user interface that is so intuitive and "easy" to use. Apple/Mac/iOS followers are almost like a cult.

2) Google invested significantly on "Material Design" because they believed that the Android interface is their brand.

3) Not too long ago, Microsoft revamped their most successful product MS Office to make it more user-friendly and that is how the Ribbon interface was born.

These are some telling examples of the importance of UX in today's digital world. UX plays a critical role in the success of games.

So, is UX that important? Let us find out.

Defining UX

Let's start with a layman's definition of UX. It is a term used as an abbreviation for User eXperience.

> *"User experience is nothing but the experience that a user would get when he/she uses or interacts with an application or a system.*

The term UX is often used along with UI (User Interface) and that too interchangeably. However, the key difference is –

> *"UI is the 'decoration' of an application or a product whereas UX is how the user feels (and hence the word experience) while using that application or product.*

Therefore, a great looking Ferrari car would be the UI and the feeling of driving something awesome is the UX!

A more complex definition of UX would be – a methodology followed while developing a user interface that would make the system (or application) more user friendly. This also ensures that the system (or application) gets a wider acceptance and there is minimal or less learning curve for usinbg the system, from an end-user's perspective.

UX has been discussed since a long time. Based on Google Trends, it is evident that there has been a continued interest in general about user experience since 2004, till date.

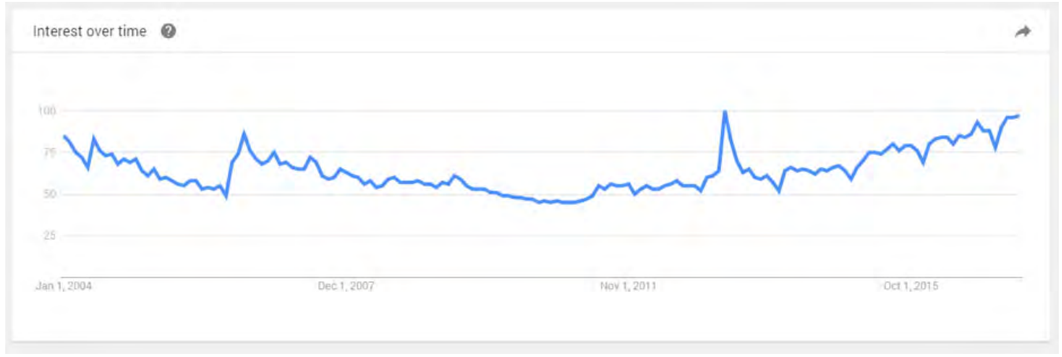


Figure 1 : UX: Interest over time

The UX Methodology

When we develop any application, we, as developers and architects, start with the technology architecture. We think about various functional and non-functional aspects of an application (or system) that we want to build and then we go about deciding the right technologies for the same.

The methodology followed for designing a user experience for an application is somewhat similar.
It is generally based on the "Five Planes" concept explained in the book '*The Elements of User Experience: User-Centered Design for the Web*', by Jesse James Garrett.

Here is a quick summary of the methodology followed:

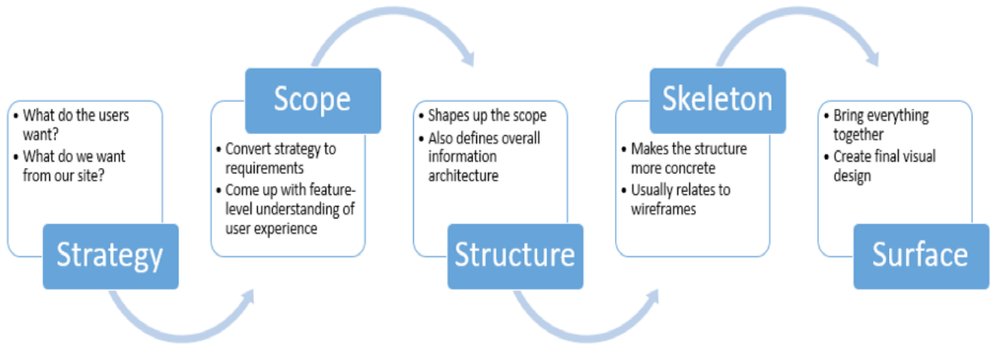


Figure 2: UX Methodology

The process above is self-explanatory. It bodes well with the agile development process in terms of multiple iterations of UX that are carried out during the agile development process.

Why UX Matters?

Some key tenets of considering UX as a key component of your product development process are:

1) UX would make the product more user-friendly because of the ease of use. This is typically critical for products or apps that are designed for end-users (B2C apps). Web apps particularly fall in this category and hence it is critical that UX is a key aspect of the web app development process.

2) UX also has a role to play in customer loyalty due to the stickiness of a product or brand that usually UX creates. This makes a whole lot of sense in "try and buy" kind of applications like ecommerce. If your UX is good, it will instantly create user stickiness for your site. You typically like one application over another because you find it very convenient while interacting with application. So for instance, you find Amazon.com better over some other ecommerce web site because you can easily find your products. Or you would find Uber better over some other cab aggregation service app, because for you, Uber is intuitive to use.

3) UX can also indirectly result into building a competitive advantage. The "experience" that users have may encourage (or force) users to continue using your app or product, over other products.

4) A good UX would result in reduced support costs as well. If users are able to find what they want in the product, then they may not be reaching out to support personnel. Hence, you would save on your support cost to certain extent if you invest in building a good UX.

The "U" (use) of UX: Deep dive

Now that we have understood the usefulness of UX in general, we will dive deep further by understanding the relevance of UX from various perspectives.

UX and Users

Well, a good clean well-designed app clearly means that the users are going to find it very easy to use it.

However, while designing the UX, you should also consider the user-base.

For a typical B2C kind of application, the user base could be from kids to senior citizens and UX needs to cater to all such scenarios. Typically, in the UX process, a lot of attention is given to the user profile. Factors that are considered are types of user, age, targeted market (local/international) and so on.

For developers or programmers (and especially for web developers), it is important to wear the hat of the consumer (or the end-user of app) for a while to understand if the application you are building is user-friendly or not.

Nowadays, end users or simply users of app/product expect the user experience to be good, else they simply move away to a similar app/product, or they continue ranting about a bad UI.

In summary, UX is certainly a must-have aspect from an end-user perspective.

Know Thy User, For He Is Not Thee.

UX Success Story

Here is an interesting story about how a small change in UX resulted in a massive upside in revenue for a major ecommerce company:

1) When the user clicked on the Checkout button, they were redirected to a registration page that had only a few fields to be filled in.

2) However, the users were not too keen on going through registration process. In addition, that was resulting into them not completing the purchase. This was a revenue loss!

3) This was found out during a UX study and the solution was to replace "Register" with "Continue" button that would make it voluntary for users to register. This ensured that users could proceed to checkout and that resulted in 45% increase in sales. This is famously called $300 Million button fix. You can read more about it here.

UX and Apps

Everything is an app now and we live in a world where we are using one app or the other – be it your smartphone, be it on-line shopping or your favorite IDE like VS Code.

Each app has different UX needs.

Your phone-calling app needs to have a simple and functional UX.

A car-racing game that you play needs to have a very catchy and vibrant design.

A trading platform that a broker uses needs to have a UX which is complex enough to cover all aspects of trading, but at the same time simple enough to have all the information at his/her fingertips.

If you are developing a web application that would be accessed by users on mobile and on web browser, then the UX needs to be responsive and usable on both form factors.

Any app that you choose to develop or work on, you need to make sure that the UX is fitting to the need of app. If it is a consumer app, then the UX needs to be catchy. If it is a business application, a catchy UI with lots of colors may look overdone. Hence, the UX needs to focus more on functional aspects in this case.

We, as developers, need to be cognizant of this aspect while developing apps.

UX and SDLC Process

Irrespective of the Software Development Life Cycle (SDLC) process you follow, UX will be a critical and important part of the development process. This is typically valid for products that are built afresh.

Things like responsive design, breadcrumbs in UI, smooth transitions, theming etc. are now a given, in any app development process. This means that the software development process needs to factor in the effort required for building user interface and teams should have people with right skill-sets in order to handle such requirements.

It is quite imperative that you as a programmer/developer should know about this basic UX aspect. Right from product manager to scrum master to a developer or tester – everyone – needs to have good understanding of UX of the app or product they are building.

OpenTable – a famous restaurant booking platform – is a good example of how they have embedded designers (and hence UX) into their SLDC process and how they are reaping the benefits of the same. The key aspects are:

1) First, they have clearly identified two different "consumers" of their service – the restaurants and the diners.

2) Then they built teams that are a cohesive unit of both designers and engineers (or programmers). Interestingly, both the teams are closely involved in design and development process together.

3) There is a significant focus on collaboration between various team members playing different roles.

You can get more details about this case study here.

UX and Cost

UX comes with a front-loaded cost. Experienced UX designers and the effort required to build UX for a product or app involves significant costs.

However, the ROI on UX is realized when because of your good UX, your app is being used by many people or your app is serving the purpose for which it was build. While the exact ROI is difficult to calculate, it is said that the cost to benefit ratio for usability is $1 : $10 to $100. You can visit this infographic for more details on similar metrics.

Better UX = Happy Customers = $$$

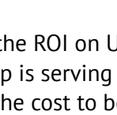
Conclusion:

It is quite evident that UX is certainly not a hype anymore.

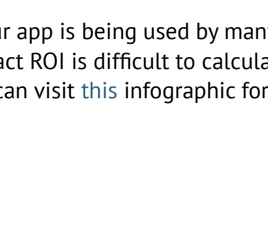
But it is also important to keep in mind that any tool that is used without the right context or is overused for the sake of using it, would not result into the desired outcome.

The case with UX is somewhat similar. You need to understand all aspects of an application "experience" and then design UX accordingly. You also need to make room for UX in your cost calculations, your execution process and well, in your own thinking process too.

Because, UX is not User eXperience anymore. It is User experience eXpected! ■



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 Suprotim Agarwal for reviewing this article.

Damir Arh

.NET STANDARD 2.0 and XAML STANDARD

At Build 2017, there were two announcements which could have a big effect on the future of cross-platform development in .NET: .NET Standard 2.0 and XAML Standard.
In this article we'll take a detailed look at what they are and why they might be important.

What is .NET Standard?

Since the original release of .NET framework in 2002, the .NET ecosystem expanded with many alternative .NET runtimes: .NET Core, Universal Windows Platform (UWP), Xamarin, Unity Game Engine and others. Although these runtimes are quite similar to each other, there are some differences between their base class libraries (BCL), which makes it difficult to share source code between projects targeting different runtimes.

Microsoft's first attempt at solving this problem was using portable class libraries (PCL), which allowed building binary compatible assemblies that could be used with multiple runtimes. Portable class libraries required the target runtimes to be selected when creating the library, as based on this choice, the intersection of available APIs was calculated.

With increasing number of runtimes and their different versions, this approach was becoming ever more difficult manage.

Hence, .NET Standard was introduced as an alternative for easier cross-platform development. It takes a reversed approach to determining the available set of APIs. Instead of calculating them based on the targeted runtimes, it specifies them on its own and requires the runtimes to support them.

For existing runtimes, it specifies several different versions with an increasing number of supported APIs.

Target Platform	1.0	1.1	1.2	1.3	1.4	1.5	1.6
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0
.NET Framework	4.5	4.5	4.5.1	4.6	4.6.1	4.6.2	4.6.3
Xamarin Platforms	*	*	*	*	*	*	*
Universal Windows Platform	10.0	10.0	10.0	10.0	10.0		
Windows Store	8.0	8.0	8.1				
Windows Phone	8.1	8.1	8.1				
Windows Phone Silverlight	8.0						

Table 1: Supported runtimes for different .NET Standard versions

To learn more about .NET Standard, you can read my previous article on DotNetCurry.com: .NET Standard – Simplifying Cross Platform Development.

.NET Standard 2.0

The biggest obstacle to a larger .NET Standard 1.x adoption was a rather small set of available APIs.

The goal of .NET Standard 2.0 is to remove this obstacle.

In comparison to .NET Standard 1.6, .NET Standard 2.0 includes more than 20,000 additional APIs. These APIs are already a part of .NET framework, but have been missing from .NET Standard before.

API Scope

Since .NET Standard is focused on cross-platform compatibility, it still doesn't include all of the .NET framework APIs and it never will.

By design, it excludes all application model APIs, such as WPF, Windows Forms, ASP.NET and others. It also excludes all Windows specific APIs that were included in .NET framework, e.g. APIs for working with registry and event logs. Most of the .NET framework APIs that do not fall in the above two categories are included in .NET Standard 2.0.

Some of the most notable additions that were missing from .NET Standard 1.6 are:

- `System.Data` namespace with `DataSet`, `DataTable`, `DataColumn` and other classes for local data manipulation that were commonly used in earlier versions of .NET framework before the release Entity Framework.

- `XmlDocument` and `XPath` based XML parsers in addition to `XDocument`, which was the only one available before .NET Standard 2.0.

- `System.Net.Mail`, `System.Net.WebSockets` and other previously missing network related APIs.

- Low level threading APIs such as `Thread` and `ThreadPool`.

There's also one namespace that remains unsupported, but is worth mentioning as it might be a bit surprising: `System.Drawing`. That's because it is mostly just a thin layer over the native GDI+ Windows component and therefore can't be easily ported to other runtimes.

Since it is probably the most commonly used .NET framework API that's still missing from .NET Standard, the .NET team is currently considering different options on how to make it available elsewhere and include it in .NET Standard. We can expect it to be added at least partially in future versions of .NET Standard.

Of course, it's impossible to list all the changes here. The best place to check for individual API availability in .NET Standard 2.0 is the official reference documentation.

Assembly Compatibility

An increased API set in .NET Standard 2.0 will make it much easier to compile existing source code that was originally written for .NET framework against .NET Standard, and make it available to other runtimes. At least as long as it is not dependent on particular application models.

This means that business logic could be shared across runtimes, but applications will still need to be written for each runtime.

However, our own source code, even with business logic, often depends on other third party libraries, not only on the base class library APIs that are now exposed via .NET Standard.

As expected, .NET Standard libraries can reference other .NET Standard libraries that target the same or lower version. They can also reference compatible portable class libraries.

Unfortunately, a large majority of third party libraries that are currently available on NuGet does not target .NET Standard or PCL, but rather the .NET framework. Although many of those could maybe easily just recompiled for .NET Standard with minimal or no code changes at all, this still needs to be done by their authors who might not actively maintain them anymore.

To avoid that requirement altogether, .NET Standard 2.0 includes a special compatibility shim, which makes it possible to reference existing .NET framework libraries from .NET Standard libraries without recompilation.

To allow that, any references to .NET framework classes from the referenced .NET framework library are type forwarded to their counterparts in .NET Standard. Hence, the compiled code will also work in other runtimes, not only in .NET framework.

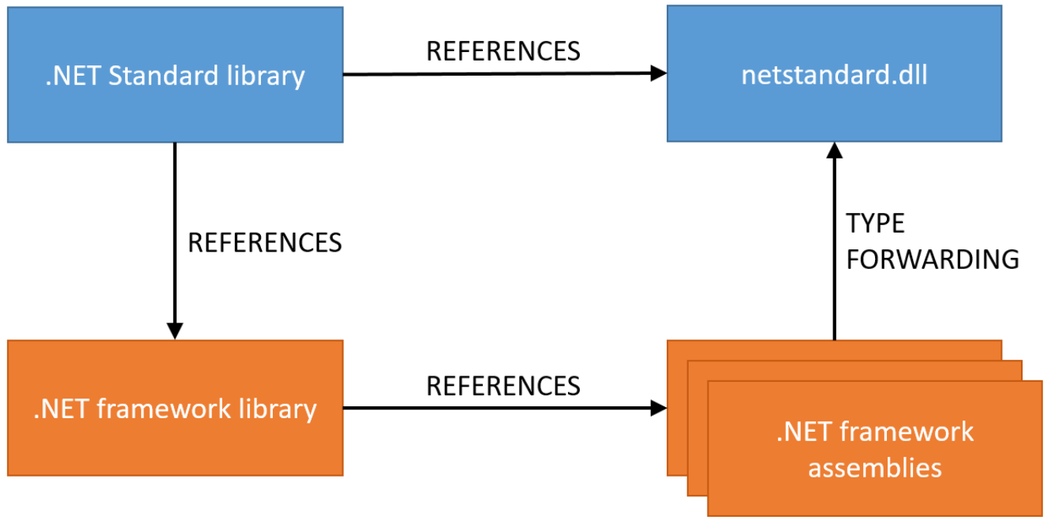


Image 1: Type forwarding for referenced .NET framework libraries

Of course, this doesn't mean that any .NET framework library on NuGet will just work with .NET Standard. If it uses any APIs that are missing from .NET standard, the call will fail.

To estimate the severity of this issue, Microsoft did an analysis of all the libraries that are currently available on NuGet and published the results. According to it, over two thirds of libraries in their latest version are API compatible with .NET Standard 2.0, i.e. they either target .NET Standard or PCL, or they target .NET framework but only use APIs that are part of .NET Standard 2.0.

Almost two thirds of those that are incompatible, depend on APIs in specific application models (WPF, ASP.NET, etc.) and are therefore not appropriate for use from .NET Standard.

The rest mostly use APIs that were already considered for .NET Standard but were excluded as too hard to implement across all runtimes. It is therefore not expected that compatibility will significantly increase with future .NET Standard versions.

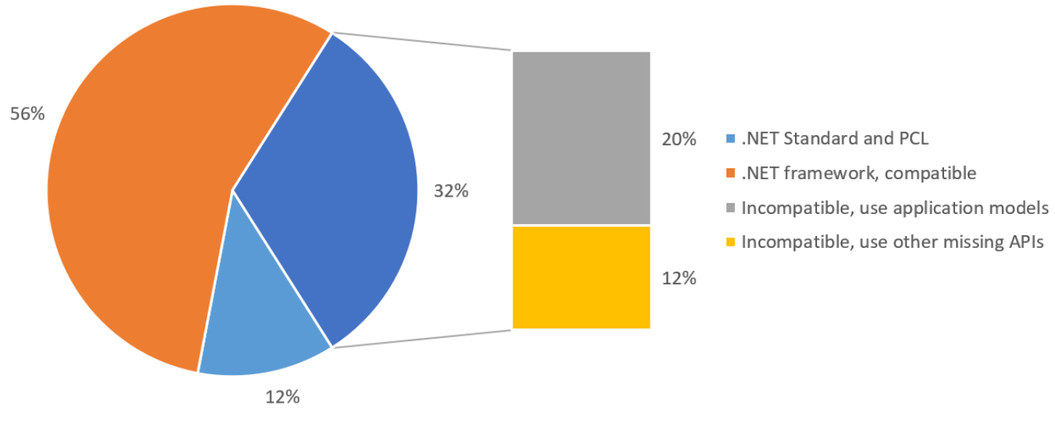


Image 2: Assembly compatibility in NuGet (source: .NET Standard – NuGet Analysis)

A small number of API compatible libraries from the above analysis (less than 7%) uses P/Invoke to call into native libraries. Although these assemblies can be used from .NET Standard, they will only work on Windows (when referenced from .NET framework or .NET Core applications) but will fail on other operating systems because of their dependencies on native binaries.

Availability

During Build 2017, .NET Core 2.0 Preview 1 was released with full support for .NET Standard 2.0. This allows applications written for .NET Core 2.0 Preview 1 to reference .NET Standard 2.0 libraries and API compatible .NET framework libraries and still run on all three currently supported operating systems: Windows, Linux and macOS.

.NET Core 2.0 Preview 1 is freely available for download, but it is not yet ready for production use. However, the SDK is installed side-by-side with the current stable version and you can test it without affecting your work on production projects for .NET Core.

To develop applications for .NET Core 2.0 and .NET Standard 2.0 you can install Visual Studio 2017 Preview 15.3. It also supports side-by-side installation with the stable version of Visual Studio 2017 and will thus not affect your regular .NET development. If you're not using Windows, you can also develop for .NET Core 2.0 using the latest version of Visual Studio for Mac or Visual Studio Code with its C# extension.

The final version of .NET Core 2.0 and .NET Standard 2.0 is planned for the third quarter of 2017. At that time, full support for .NET Standard 2.0 should also be available in .NET framework and Xamarin.

Universal Windows Platform (UWP) will add support for .NET Standard 2.0 with Windows 10 Fall Creators Update, which is announced for fall 2017. Microsoft is also collaborating with Unity Game Engine developers to add .NET Standard 2.0 support as part of their transition to a newer version of Mono. No information about its availability has been published yet.

XAML Standard

As a complement to .NET Standard, XAML Standard was also announced at Build 2017. It is an effort to define standard XAML vocabulary elements for describing user interfaces. The current proposal is being designed publicly on GitHub.

The first version of the standard (XAML Standard 1.0) is planned for release later this year.
After the specification is completed, support is planned to be added to Universal Windows Platform and Xamarin applications. This will allow views to be shared between the applications for these two runtimes. Additional XAML-based application models might add support for XAML Standard in the future.

Conclusion:

With version 2.0, .NET Standard is evolving from a promising tool for cross-platform development into an essential part of making .NET Core a more viable and attractive runtime. The larger set of APIs and the ability to use existing .NET framework libraries will make it much easier to create cross-platform libraries for sharing common business logic between multiple runtimes (e.g. an ASP.NET Core web application, a desktop .NET framework application and a mobile Xamarin application will all share the same assemblies with business logic).

Although the tools are not yet ready for production, they can already be used for testing how well this approach can work for your projects.

XAML Standard is the UI equivalent of .NET Standard. However, it is still in very early stages and it's therefore difficult to judge how it will turn out. If it is of interest to you, now is the right time to get involved on GitHub and affect its development ■

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 Suprotim Agarwal for reviewing this article.