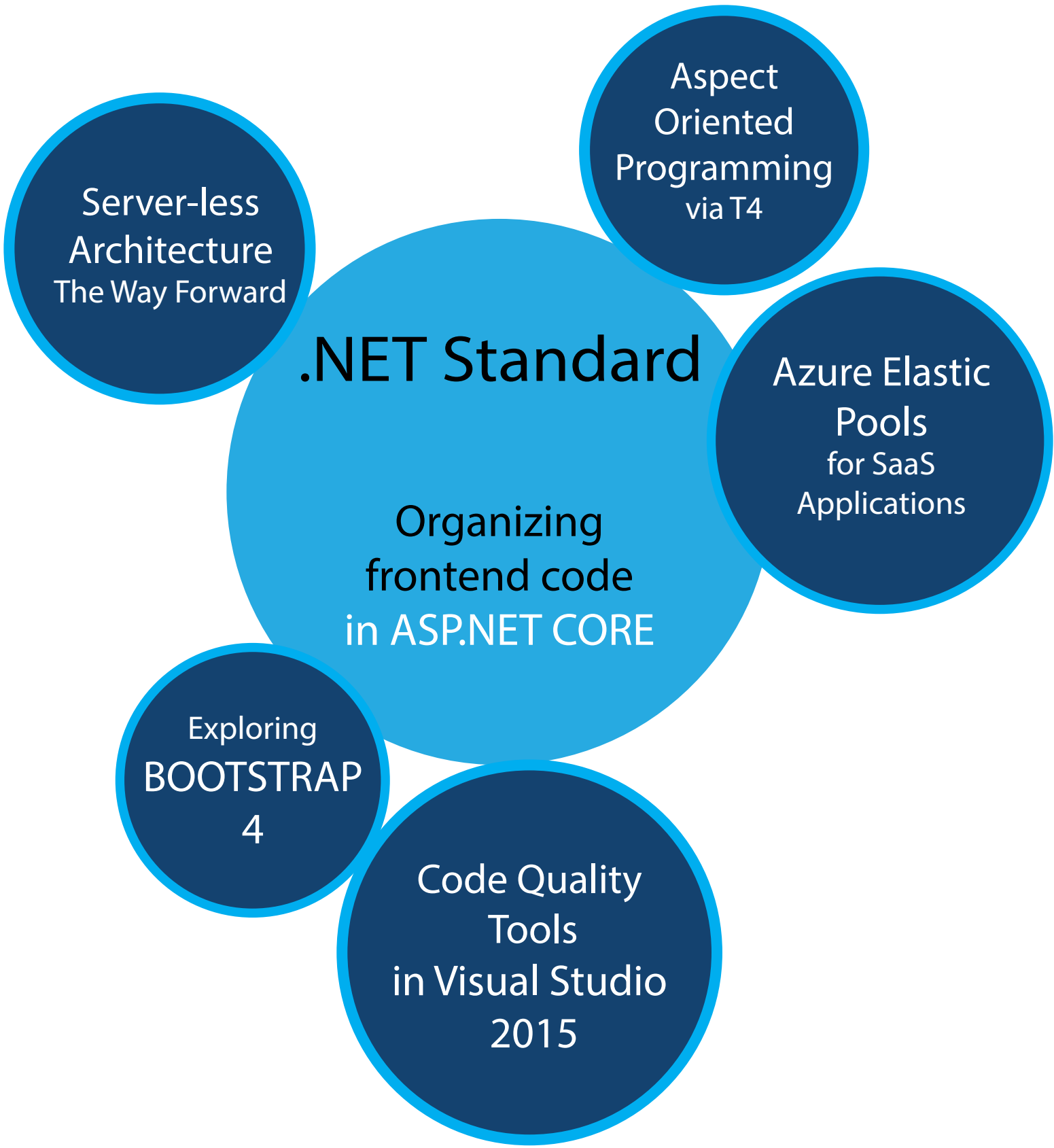
Issue 27 | Nov-Dec 2016

DNCMagazine

www.dotnetcurry.com



EDITORIAL

Editor in Chief

Dear Friends,

We are back again with the 27th Edition of the DotNetCurry (DNC) magazine.

For this edition, we have a bouquet of exclusive articles for you covering AOP Programming, .NET Standard, ASP.NET Core, Bootstrap 4, Server-less architecture, Azure Elastic Pools and Code Quality Tools.

Make sure to reach out to me directly with your comments and feedback on twitter @dotnetcurry or email me at suprotimagarwal@dotnetcurry.com

Suprotim Agarwal

THE TEAM

Editor In Chief
Suprotim Agarwal
suprotimagarwal@a2zknowledgevisuals.com

Art Director
Minal Agarwal

Contributing Authors
Benjamin Jakobus
Damir Arh
Daniel Jimenez Garcia
Gouri Sohoni
Rahul Sahasrabuddhe
Vikram Pendse
Yacoub Massad

Technical Reviewers
Damir Arh
Kunal Chandratre
Suprotim Agarwal

Next Edition
January 2017

CONTENTS

ASP.NET MVC CONTROLS

WORK EFFORTLESSLY WITH ASP.NET MVC

Quickly create advanced, stylish, and high performing UIs for ASP.NET MVC with Ignite UI MVC. Leverage the full power of Infragistics' JavaScript-based jQuery UI/HTML5 control suite with easy-to-use ASP.NET MVC helpers and get a jump start on even the most demanding Web applications.

Download ASP.NET MVC Controls as part of the Ultimate Developer toolkit.

DOWNLOAD FREE TRIAL

INFRAGISTICS®

Damir Arh

Microsoft finally released version 1.0 of .NET Core in June 2016. .NET Core is a cross platform open source reimplementation of .NET Framework for the modern day. In spite of the name changes in the past, it seems recognizable enough in the developer community. One could hardly say that about .NET Standard, which is often mentioned in connection to it. Not many developers would dare to explain it with confidence. In this article, I will try to describe where it originates from, and why I think it is important for developers to know about it.

.NET Standard - Simplifying Cross Platform Development

The Full .NET Framework

The first version of .NET framework was released in 2002. At the beginning of August 2016 it reached version 4.6.2. In more than 14 years since its initial release, there were a lot of improvements, but many core principles remained unchanged. Most importantly, it is only available for Windows operating system, its desktop version to be precise. Windows even comes with a version of .NET framework, preinstalled. However, .NET framework updates, are independent of the operating system.

At the core of .NET framework, there is the Common Language Runtime (CLR), a virtual machine executing the common Intermediate language (IL). IL is a replacement for platform and architecture specific machine code. On top of CLR is the Base Class Library (BCL) consisting of the base classes, and types. BCL is a part of the larger .NET Framework Class Library (FCL), which additionally includes the higher-level application frameworks, such as ASP.NET, WPF, LINQ, and others.

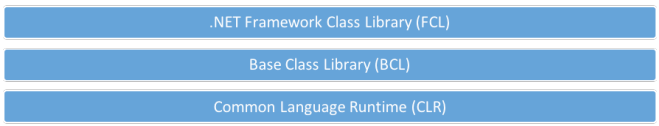


Image 1: .NET Framework Stack

In theory, this structure of .NET framework allows its executables to run on different platforms, as long as they all provide a CLR implementation, and have the same set of class libraries available for those applications. Unfortunately, it is a bit more complicated than that.

Alternative .NET Runtimes

In 2007, Microsoft released the first version of Silverlight, an application framework similar to Adobe Flash for development of rich internet applications that could run inside a browser with a special plugin installed. The last major version of the framework, Silverlight 5, was released in 2011. Active development stopped after that because browser plugins started losing traction in favor of HTML5 and JavaScript applications.

Development of Silverlight applications was similar to development of .NET framework applications: the same languages could be used (C# and Visual Basic .NET), and a subset of .NET framework class libraries was available for it. The UI framework was XAML based (like WPF), but there were enough differences between the two to prevent sharing the same code. In short, Silverlight was a new independent .NET runtime, which had enough in common with .NET framework that existing .NET developers could learn it easily; but was different enough to make porting of existing applications difficult.

The trend continued with Windows Phone 7 in 2010. Its application framework was strongly based on Silverlight, yet was different enough to address the specifics of the new mobile platform. The platform evolved with new Windows Phone operating system versions until Windows Phone 8.1.

At that time, an alternative application framework for Windows Phone devices became available. It was based on WinRT (also known as Windows Runtime) - another spin-off of Silverlight, designed for development of Windows Store application for Windows 8, and its short-lived Windows RT version for ARM devices. This was Microsoft's first attempt at a universal application platform for Windows desktop and phone devices.

With Windows 10, this runtime evolved into the Universal Windows Platform (UWP), a common application framework for all flavors of Windows 10 operating system and devices it runs on: desktop, mobile, IoT Core, Xbox One, HoloLens and Surface Hub.

The latest .NET runtime created by Microsoft was .NET Core. It is completely open source and currently available for Windows, Linux and macOS. UWP is actually running on top of it, but unlike .NET Core itself, it is not cross-platform. The only two cross-platform application models for .NET Core are console applications and ASP.NET Core. Hopefully, more will be available in the future.

Outside Microsoft, the Mono runtime is in active development in parallel to .the NET framework even before .NET framework was originally released 1.0 in 2002. It is an independent .NET framework compatible runtime with support for many different platforms: Windows, Linux, BSD, all Apple operating systems, Android and even gaming consoles Sony PlayStation 3 and 4, and Nintendo Wii. It also serves as the basis for two other renowned products:

- Unity 3D game engine uses Mono runtime as the basis for its cross platform support, and allows its developers to use C# and a subset of .NET libraries for all the scripting in their games.

- Xamarin bases its Xamarin.iOS, Xamarin.Android and Xamarin.Mac products on Mono runtime for cross platform development using C# and .NET libraries.

In March 2016, Microsoft acquired Xamarin and became the owner of their products. In addition, both Unity and Mono are now part of .NET foundation, an independent organization dedicated to strengthening the .NET ecosystem. Hence, Microsoft is now more closely involved in development of all .NET runtimes than it has ever been before.

Challenges of Cross Platform Development

Due to the specifics of individual platforms, not all .NET applications can be packaged the same way. Some can be distributed as pure assemblies in IL, while others need to be wrapped into single platform specific packages or even precompiled into machine code. However, this is not the main obstacle for effective cross platform development in .NET. The build process can take care of that easily enough.

A much bigger problem is the differences in class libraries that are available for individual .NET runtimes. This prevents the source code to be the same for all of them. There are two common approaches to cope with that problem:

- The code that needs to be different because of the differences in class libraries can be refactored into separate classes. For each runtime, these classes can then be implemented differently while the rest of the code can remain identical for all targeted runtimes.

- Where this approach is too granular, conditional compilation allows individual lines of code to differ for different runtimes in the same source code file. By defining different conditional compilation symbols for different runtimes, only the correct lines of code will be compiled for each runtime.

```
#if NETFX_CORE
  return "Windows Store application";
#elif WINDOWSPHONE_APP
  return "Windows Phone application";
#endif
```

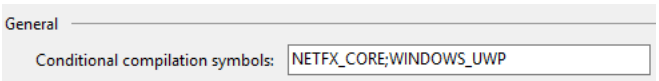


Image 2: Conditional compilation symbols in project build properties

Portable Class Libraries

In 2011, Microsoft introduced Portable Class Libraries (PCL) to make cross platform development easier. The basic idea behind them was to define a common set of APIs that are available in all of the targeted runtimes. For this purpose, special contract assemblies were defined that could be referenced from PCL source code. At runtime, the calls to classes in those contract assemblies, are forwarded to the assemblies actually implementing those classes in each runtime. With this approach, the same PCL binaries work with different runtimes even if the classes are not implemented in the same assemblies in all of them, as long as they behave the same.

The first step in creating a portable class library involves selecting the exact runtimes, and their version that the library will target. Based on the

selection, the common set of APIs across all of them is determined, and made available for use. Of course, the more runtimes the library targets, the smaller is the set of common APIs available.

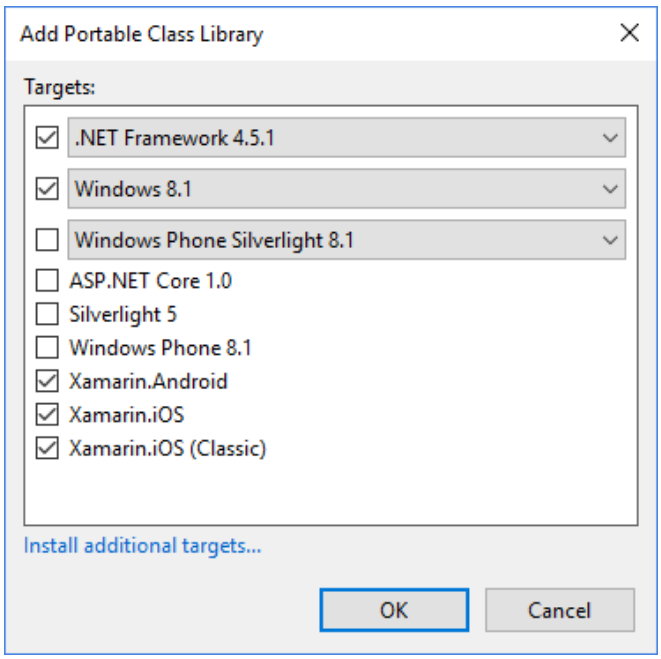


Image 3: Targeted runtimes for a portable class library

This can help a lot in the following two scenarios:

- When developing a cross-platform application, common code can be put in a separate portable class library that only needs to be compiled once, and can then be referenced from other assemblies that target one of its selected target platforms. There is no need any more to have a different class library project for each platform and include the same source code files in these multiple platform specific projects.

- Class library authors can build a single PCL assembly for all the platforms they want to target, and distribute it as a NuGet package that can be consumed on any supported platform.

There is a disadvantage to this approach, though. Only after explicitly selecting the target platforms and their versions in advance, the intersection of the available APIs for use in the class library is determined. This makes it impossible to target a newly released platform, or a new version of an existing platform, until all the tooling is updated. Even if its set of APIs matches one of the existing ones, developers must explicitly select the new target to support it. Existing libraries must also be recompiled for the new target, even if no source code change is required. All of this brings additional complexity and means more work for library authors.

.NET Standard

.NET Standard is building on the basic concepts of portable class libraries, while trying to resolve their known issues. Instead of determining the common API for the targeted platforms, it defines a set of APIs independently of them. There are already multiple versions of .NET Standard defined. A higher version of standard is backward compatible with all previous versions. It only adds additional APIs to its previous version. Each existing platform supports .NET Standard up until a specific version:

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: Minimum target platform version with support for each .NET Standard version

Class library authors can now select a .NET standard version to target with their library. A library targeting a specific .NET Standard version will work on any platform implementing this version of the standard. If a platform is updated to support a higher version of .NET Standard, or if a new platform is released, all existing libraries targeting the newly supported version of .NET Standard will automatically work with it without any changes required by their respective authors. Tooling will also only need to be updated when a new version of .NET Standard is released.

At the time of this writing, the process of creating a .NET Standard library in Visual Studio is still a bit

unintuitive. You start by creating a new Portable Class Library project. It does not matter which targets you choose to support, as you will change that in the next step any way. When the new project is created, you need to open the project properties. At the bottom of the *Library* page, under the list of selected platforms there is a link to *Target .NET Platform Standard*.

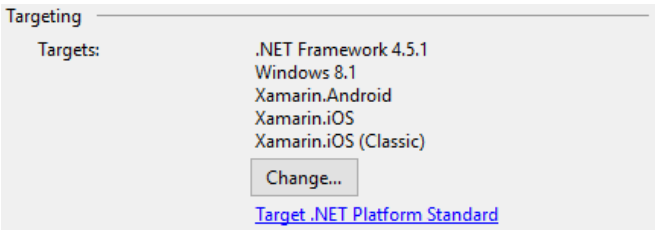


Image 4: Target .NET Platform Standard

When you click on the link, a confirmation dialog will be displayed, warning you that you are about to change the library target which will result in a different set of available APIs.

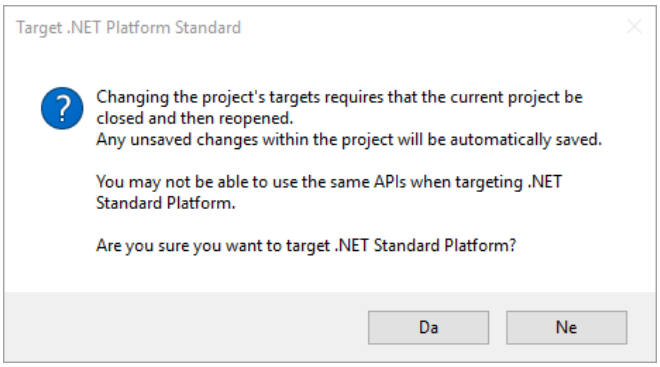


Image 5: Confirmation dialog

Confirm the dialog and wait for the project to reload. The *Targeting* section of the *Library* page in project properties has now changed. You have the option to select a version of .NET Standard to target, along with a link to switch the targeting back to portable class library profiles.

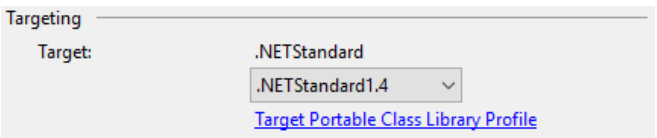


Image 6: Select .NET Standard version to target

Depending on the APIs that you need, you can now select the version to target and start writing code. You can reference the created library from any other project targeting a compatible platform. Since I selected .NET Standard 1.4 in the screenshot above, I can reference my library from .NET Core projects, from UWP projects, and from classic Windows desktop projects for .NET framework 4.6.1 or above. If both projects are in the same solution, I can also add a reference to the project, not only to the compiled assembly. Just like any other cross project reference in Visual Studio, I can add it by checking the checkbox in front of the project in the *Reference Manager* dialog.

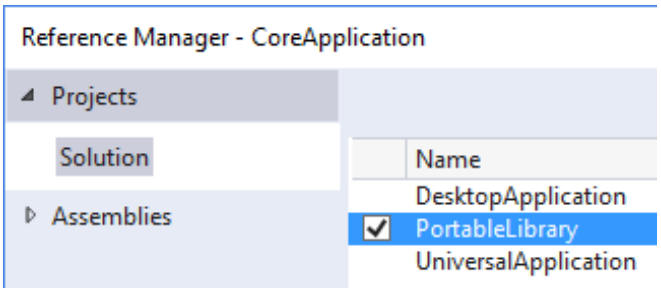


Image 7: Add reference to .NET Standard library project

If I try to reference it from an incompatible project (e.g. if I wanted to reference it from a .NET framework 4.6 project, which does not implement .NET Standard 1.4), Visual Studio would not allow it and would display an incompatibility warning instead.

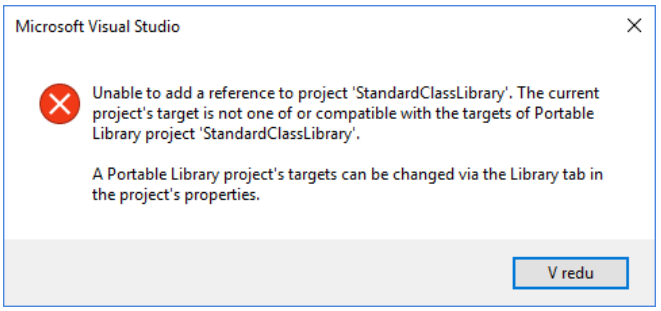


Image 8: .NET Standard library cannot be referenced from an incompatible project

NuGet version 3.4 and above also supports .NET Standard libraries, i.e. class libraries can be packaged as targeting a version of .NET Standard, and can be consumed in other projects as such. This should make it possible for library authors to start targeting .NET Standard instead of PCL profiles, at least for the majority of PCL profiles, which have compatible .NET Standard equivalent.

.NET Standard 2.0

In its current state, .NET Standard is a cleaned up version of Portable Class Libraries. It has improved usability, but its set of available APIs is similarly limited. This prevents many of the existing class libraries to be ported to .NET Standard and hence becoming available on other platforms. With the introduction of .NET Core, cross-platform development is becoming even more important, therefore it is the primary focus in the plans for .NET Standard 2.0.

.NET Standard 2.0 will initially be supported by .NET framework, the Xamarin platforms and .NET Core (this includes UWP, which is implemented on top of it). The set of APIs in .NET Standard 2.0 should be much larger than the existing set in .NET Standard 1.6. According to the current plan, all APIs that are present in both .NET framework and in Xamarin platforms, will be added to .NET Core 1.1. These APIs will also serve as candidates for adding to .NET Standard 2.0:

- Those that can be implemented on all platforms will be added to .NET Standard 2.0 as required APIs.

- Those that are platform specific or legacy, will be added to .NET Standard 2.0 as optional APIs.

Optional APIs will be made possible by the fact that just like PCL assemblies, .NET Standard assemblies also only do the type forwarding to the assemblies actually implementing the types. Implementations of optional APIs will not be a part of .NET Standard library. Instead, separate NuGet packages with their implementation will be available for supported platforms.

If such optional APIs will not be used in a project, additional NuGet packages will not need to be installed. However, as soon as an API will be used for the first time, the build will fail without the corresponding NuGet package installed. On platforms with support for the API, the issue will be resolved by installing the appropriate NuGet package. Visual Studio tooling should make the process of installing the right NuGet packages, easier. On non-supported platforms, the API will not be available for use. This approach should make it easier to develop for a subset of platforms with common optional APIs, e.g. to use Windows registry APIs in applications for .NET framework and .NET Core on Windows.

With a larger set of APIs available, it should become easier to port existing class libraries to .NET Standard. However, class libraries often depend on other class libraries, making them difficult to port until all of their dependencies are ported as well. To alleviate this problem, .NET Standard libraries will not be limited to referencing other .NET Standard libraries. Additionally, they will also be able to reference portable class libraries targeting compatible PCL profiles, and even .NET framework libraries that only use APIs included in .NET Standard 2.0.

To make consuming of .NET Standard libraries more convenient, a breaking change has been announced: .NET Standard 2.0 will be compatible with .NET Standard 1.4, but not with .NET Standard 1.5 and 1.6. APIs that were added in these two versions will be missing in .NET Standard 2.0, and will only be added back in later versions. The main reason for this decision is to make .NET Standard compatible with .NET framework 4.6.1, which does not implement the APIs that were added in.NET Standard 1.5 or 1.6. Since .NET framework is installed system-wide and cannot be bundled with applications, most computers will only have .NET framework 4.6.1 installed, because this version has already been distributed with a Windows 10 update. Developers who will therefore want to target this version, would not be able to use .NET Standard 2.0 libraries without this breaking change.

Conclusion:

Understanding the details and inner workings of .NET Standard is most important to library authors and developers of cross platform applications. Both groups should already seriously consider changing their class libraries to target .NET Standard, instead of PCL profiles, where ever possible. Considering the fact that .NET Standard 2.0 will not be compatible with existing .NET Standard versions above 1.4, it makes most sense to target .NET Standard 1.4 until

.NET Standard 2.0 is released.

Developers of applications that only target a single platform do not need to concern themselves with .NET Standard, just as they did not need to know about portable class libraries in the past. They might only see the term when checking compatibility of libraries they want to use, and even in this case, tooling should be able to protect them from implementation details.

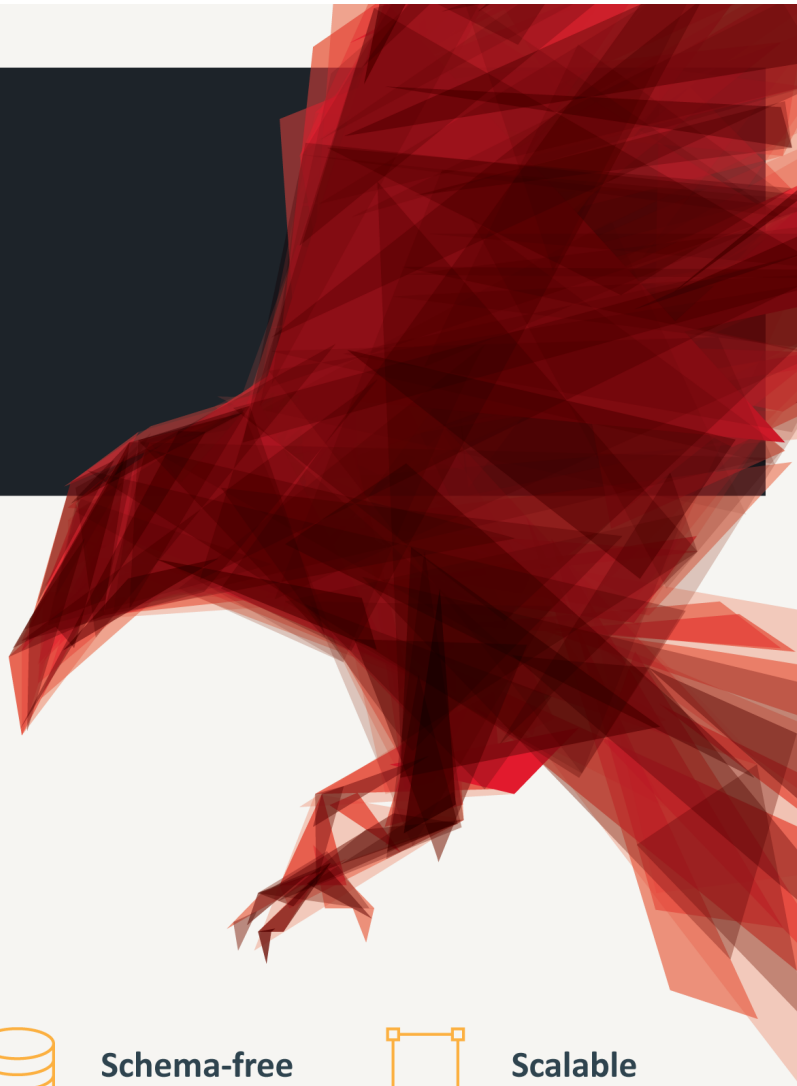
In its current state, .NET Standard appears to be a promising successor to portable class libraries. It will show its full potential if .NET Standard 2.0 manages to deliver on all its promises ■

About the Author

damir arh

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.

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

Yacoub Massad

Aspect Oriented Programming via T4

This article discusses with examples how we can use the Text Template Transformation Toolkit (T4) to create aspects.

Introduction

Aspect Oriented Programming (AOP) allows us to remove code duplication and code tangling that we get when we address cross-cutting concerns (e.g. logging). This allows us to focus on the real problem that we are trying to solve, as supporting functions and logic for cross-cutting concerns are isolated. In the article Aspect Oriented Programming in C# with SOLID (bit.ly/dnc-aop-solid), I discussed some options for doing AOP in SOLID code bases. For more details on what AOP is, and the problems we face when we do AOP in SOLID code bases, please refer to that article.

One of the options for doing AOP is via Text Template Transformation Toolkit a.k.a. T4. This article will discuss T4 in more details, provide examples for creating a retry aspect as well as a logging aspect via T4, and discuss the benefits of such an approach. The example code is available on GitHub at http://bit.ly/dnc-m27-aopviat4.

Text Template Transformation Toolkit (T4)

The Text Template Transformation Toolkit (T4) is a framework in Visual Studio that allows developers to generate text, based on templates. Developers write templates that can contain static text, and also code blocks that allow for the generation of dynamic text. Consider the following T4 example:

```
<People>
<#
  for (int i = 20; i < 30; i++)
  {
#>
    <Person Age="<#= i #>"/>
<#
  }
#>
</People>
```

I have highlighted the static text in the template. The other part of the template is code written in C# that controls the dynamic generation of text. This code has to be inside a control block. In this example, we use two types of control blocks.

The first type is the *standard control block*. The code in this block is surrounded by the <# and #> brackets, and it allows us to write code statements.

The second type is the *expression control block*. The code in this block is surrounded by the <#= and #> brackets, and it allows us to specify a single expression.

This particular template will generate an XML document that looks like this:

```
<People>
  <Person Age="20"/>
  <Person Age="21"/>
  <Person Age="22"/>
  <Person Age="23"/>
  <Person Age="24"/>
  <Person Age="25"/>
  <Person Age="26"/>
  <Person Age="27"/>
  <Person Age="28"/>
  <Person Age="29"/>
</People>
```

Although the <Person> tag is static in the template, it is surrounded by a standard control block that loops from 20 to 29. This will generate the static text multiple times. Notice how we have the <#= i #> expression control block to vary the value of the *Age* attribute in each iteration.

There is a third type of control block too (which we aren't using in this example) called the *class feature control block*. This code block has the <#+ and #> brackets and allows us to write methods, fields, properties and even new interfaces and classes inside it.

T4 supports two kinds of templates; runtime templates and design time templates.

Runtime templates allow the generation of text during the execution of our applications. For example, one could use a runtime template to generate some report based on data from the database.

Design time templates allows the generation of text during design time. For example, one could use a design-time template to generate some C# code for a logging decorator for some class at design time.

Since in this article we are interested to use T4 to create aspects at design time, we are going to use design time templates.

This article is about AOP via T4, and not an introduction to T4. Further information about T4 can be obtained from this article on MSDN: Code Generation and T4 Text Templates (bit.ly/dnc-t4-msdn). I will be explaining more about relevant T4 features in later sections.

The examples on GitHub

I have created C# examples on GitHub to demonstrate the ideas in this article. The solution contains 5 projects:

Project	Description
T4Aspects	This project contains RetryAspect.tt, LoggingAspect.tt, and other .tt files. These two files represent the Retry, and the Logging aspects respectively. These aspects are generic and do not apply to specific types. Many applications can use these aspects to apply them to their own types.
Library	Contains the IDocumentSource interface and a sample implementation of this interface, the DocumentSource class. This represents a library where one would put the application code.
App	This is the project that contains the Composition Root of the application. This is where the different classes are wired together. This is also where we apply the aspects to objects. This project also contains the RetryAspectLocal.tt and LoggingAspectLocal.tt files. These files are not the real aspects, instead they consume the aspect files found in the T4Aspects project to generate the decorators relevant to the application. Under these files, you can find RetryAspectLocal.cs and LoggingAspectLocal.cs respectively. These files contain the decorators generated by the T4 engine. Make sure that this project is set as the startup project when you want to run the application.
LoggingAOP	This project contains some attributes that can be used to decorate methods and parameters so that the logging aspect knows which data needs to be logged.
Logging	This project contains the ILogger interface and the basic ConsoleLogger implementation.

A simple example: a retry aspect

Sometimes, we want a certain operation to retry in case of failure. We can create an aspect that causes method invocations to retry if an exception is thrown. One way to do this is to create a T4 template that generates decorators for our interfaces that retries the execution of the decorated methods.

Let's consider the following interface that represents a document source from which we can obtain documents of some format:

```
public interface IDocumentSource
{
    Document[] GetDocuments(string
format);
}
```

We can manually write a decorator for this interface to support retrying the `GetDocuments()` method like this:

```
public class
DocumentSourceRetryDecorator :
IDocumentSource
{
 private readonly IDocumentSource
 decorated;
 private readonly int numberOfRetries;
 private readonly TimeSpan
 waitTimeBetweenRetries;
 //...
 public Document[] GetDocuments(string
 format)
 {
  int retries = 0;

  while(true)
  {
   try
   {
    return decorated.
    GetDocuments(format);
   }
   catch
   {
    retries++;

    if(retries == numberOfRetries)
    throw;
```

```
            Thread.
            Sleep(waitTimeBetweenRetries);
        }
    }
  }
}
```

However, we don't want to repeat the same thing for all interfaces in our code base. Instead, we create a T4 template that can generate a similar decorator for any number of interfaces that we like.
Let's start by considering what would change in this code for different interfaces:

1. The class name needs to change. More specifically, "DocumentSource" at the beginning of the class name needs to change.

2. The implemented interface name (after the column) needs to change.

3. The type of the "decorated" field needs to change.

4. The method signature (return type, method name, and parameters) needs to change.

5. The method call inside the try block needs to change to reflect the method signature. If the target method has no return value (its return type is void), then the "return" keyword before the method call should be removed, and instead, a "return;" statement should be put after the method call.

6. If the other interface has multiple methods, we need to generate a method in the decorator for each of the methods in the interface.
We have basically identified the pieces that need to be generated dynamically. The rest of the code is static and doesn't change from interface to interface.

Let's discuss the retry aspect example. Please take a look at the RetryAspect.tt file. At the top of the file, we have some T4 directives. T4 directives provide instructions to the T4 engine. For example,

- the **assembly** directive in the first line informs the engine to load the EnvDTE assembly because the code inside the template is going to consume types inside that assembly.
- the **import** directive works like the using directive in C# in that it allows us to reference certain types in some namespace without typing the full name of the type.
- the **include** directive allows us to include the content of another .tt file inside the current template file. This allows us to create helper methods or classes and use them from multiple templates.

Please note that these .tt files that contain helper methods/classes are usually not meant to be used directly by the T4 engine to generate text/code. Therefore, it is best to let Visual Studio know never to execute the T4 engine on these files. To do so, we can click on the .tt file, and then in the properties window clear the value of the Custom Tool property. For example, the RetryAspect.tt file is not meant to be executed directly by the T4 engine. As you will see later in this article, a template file called RetryAspectLocal.tt that includes RetryAspect.tt is the template that the T4 engine will execute.

For more information on T4 directives, you might want to check the T4 Text Template Directives (bit.ly/dnc-t4dir-msdn) reference from MSDN.

The Visual Studio Automation Object Model

The first two assemblies referenced from our template are EnvDTE and EnvDTE80. These assemblies contain types that allow us to interact with Visual Studio. We will use this in our aspect to search for some interfaces in the current solution to create the decorators for. We will be able to enumerate methods inside any interface, get the signature of these methods, and read any attributes on the methods if we want.

This looks a lot like Reflection. However, the difference is that Reflection works only with code that is compiled into assemblies, while the Visual Studio Automation Object Model can work with code in the open solution in Visual Studio, even if it is not yet compiled.

The GenerateRetryDecorator..() method

Let us continue exploring the RetryAspect.tt file. After the T4 directives, we have a class feature control block that contains a C# method called `GenerateRetryDecoratorForInterface`. This method takes an object of type `CodeInterface2` (which is a type from the EnvDTE80 assembly) and generates a retry decorator for the interface represented by this object. We use this object to obtain information about this interface like its full name (e.g. `Library.IDocumentSource`), and the namespace where it resides. We also generate the name of the decorator that we want to create. The `FormatInterfaceName` method (found in the same file) removes the "I" from the interface name so that it becomes "prettier" when we use it to generate the name of the decorator.

Next, we have some static text that represents the retry decorator that will be generated. Notice how we use expression code blocks (the ones surrounded by <#= and #>) to insert some dynamic content (e.g. the namespace, the name of the decorator, and the implemented interface) between the static text.

We then have a class feature control block that calls the `GenerateMethodsForInterface` method. This method is going to generate the methods of the decorator. We use the PushIndent and PopIndent T4 methods to control the indentation of the generated code. We pass eight spaces to the `PushIndent` method so that any code generated inside the `GenerateMethodsForInterface` method is pushed eight spaces to the right.

Before we move on to the `GenerateMethodsForInterface` method, note that after the decorator class is generated, we generate another static class whose name starts with `ExtentionMethodsForRetryAspectFor` and ends with the formatted interface name. This static class contains a single extension method that allows us to decorate objects in a more readable way. Take a look at the Composition Root of the application to see how the generated `ApplyRetryAspect` method is used.

The GenerateMethodsForInterface method

This method will loop through the methods of the interface, and for each one, it will generate a method inside the decorator. As we did before, we first obtain some information about each method from inside the control block. For example, we determine whether the method has a "void" return type, and we store this information in the *isVoid* variable. Then we have the code of the decorator method as static text that includes some control blocks to dynamically generate code relevant to the current method. For example, note how the method call inside the try block is generated differently based on the *isVoid* variable.

The RetryAspectLocal.tt file

This file lives inside the App project. It searches for some interfaces (currently the `IDocumentSource` interface only) using the `InterfaceFinder.FindInterfacesInSolution` helper method (found in the InterfaceFinder.tt file in the T4Aspects project). It then loops through the interfaces found and invokes the `GenerateRetryDecoratorForInterface` method from the RetryAspect.tt file to generate a retry decorator for each interface.

To see the power of what we have done so far, let's create a new interface inside the Library project, and then have T4 generate a retry decorator for the new interface.

1. Create the following interface inside the Library project (feel free to create any other interface if you want):

```
public interface IDocumentProcessor
{
  void ProcessDocument(Document
  document);
}
```

2. In the RetryAspectLocal.tt file, include "IDocumentProcessor" in the list of interfaces to search for like this:

```
var interfaces = InterfaceFinder.
```

```
FindInterfacesInSolution(dte,
new[] {"IDocumentSource",
"IDocumentProcessor"});
```

3. Right-click the RetryAspectLocal.tt file in the Solution Explorer window, and click Run Custom Tool. This will run the T4 engine and generate the decorators.

4. Go to the RetryAspectLocal.cs file to see the generated `DocumentProcessorRetryDecorator` class and the `ExtentionMethodsForRetryAspectForDocumentProcessor` class.

A challenge for the reader

For asynchronous methods that return Task or Task<TResult>, we need to await the tasks returned from the decorated methods. Also, it makes sense to asynchronously wait (before retrying to execute the method again) via the Task.Delay method instead of using Thread.Sleep.

What changes need to be made to support such asynchronous methods?

A more advanced example: a logging aspect

The example solution also contains a logging aspect. The purpose of this aspect is to separate the logging logic of any class into its own decorator. To see how the end result would look like, take a look at the `DocumentSourceLoggingDecorator` class in the generated LoggingAspectLocal.cs file.

As with the retry aspect, the logging aspect lives inside the T4Aspects project in the LoggingAspect.tt file. However, there are some dependencies for this aspect. Take a look at the `GenerateLoggingDecoratorForClass` method inside the LoggingAspect.tt file. It takes in a `CodeClass2` object representing the class that we are going to generate the decorator for, a `CodeInterface` object representing the interface that will be implemented by the decorator object, a `IPreInvocationLoggingDataCodeGenerator` object, and a `IPostInvocationLoggingDataCodeGenerator` object.

The last two parameters for this method represent seams that allow us to customize the logging aspect. The first one of these two parameters represent an object that knows how to generate logging data code prior to executing the decorated methods.

To explain this, let's first take a look at the `IPreInvocationLoggingDataCodeGenerator` interface in the LoggingAspect.Core.tt file. This interface has a method called `Extract` that takes in a `CodeFunction` object that represents a method, and returns an array of `LoggingDataCode` objects that contain the code (as strings) that knows how to access the name and value of the data to be logged.

For an example implementation of this interface, let's look at the LoggingDataCodeGeneratorForArgumentsBasedOnTheLogAttribute class. This class extracts logging data code based on the Log attribute that developers can use to decorate method parameters in their classes. Take a look at the `GetDocuments` method in the DocumentSource class:

```
public Document[]
GetDocuments([Log("Document Format")]
string format)

{
  return
  Enumerable
   .Range(0, 10)
   .Select(x => new Document("document"
   + x, "content" + x))
   .ToArray();
}
```

The *format* parameter is decorated with the Log attribute. The `LoggingDataCodeGeneratorForArgumentsBasedOnTheLogAttribute` class would in this case generate a LoggingDataCode object that contains the following code (as strings):

NameCode: "Document Format"

ValueCode: format

When a decorator is generated (in LoggingAspectLocal.cs), the following code will be included in the generated class as a result of this:

```
new LoggingData{ Name = "Document
Format", Value = format},
```

So the content of `NameCode` is actually C# code that will be put after "Name = ", and the content of `ValueCode` is C# code that would be put after "Value = ".

The other `LoggingDataCode` generator classes work in a similar way. Here is a list of these classes:

LoggingDataCodeGeneratorFor ArgumentsBasedOn TheLogAttribute,
LoggingDataCodeGeneratorForArgumentsBasedOn TheLogCountAttribute,
LoggingDataCodeGeneratorForReturnValueBasedOn TheLogAttribute,
LoggingDataCodeGeneratorForReturnValueBasedOn TheLogCountAttribute,
MethodDescriptionLoggingDataCodeGenerator.

The structure of the rest of the code in LoggingAspect.tt is somewhat similar to that of RetryAspect.tt. We basically loop through each of the methods and we have a mix of static code and control blocks to generate a try/catch block that invokes the decorated method. If there is an error, we invoke the `LogError` method on the ILogger dependency (in the decorator class). If on the other hand, the call is successful, we invoke `LogSuccess`.

Notice how we use the `LoggingDataCode` generator dependencies to extract the `preInvocationLoggingDataCodeList` list that contains `LoggingDataCode` items for pre-invocation data like arguments, and the `postInvocationLoggingDataCodeList` list that contains `LoggingDataCode` items for the return value.

As we did with the retry aspect, feel free to:

- create new interfaces and classes in the Library project
- decorate the class methods with Log, LogCount, and MethodDescription attributes
- inform our aspect system about them inside LoggingAspectLocal.tt
- see the logging decorator generated for them in LoggingAspectLocal.cs
- then use the decorators inside the Composition Root to see the logged messages printed to the console.

T4 versus Reflection

In the previous article, we used Reflection at runtime to extract logging data. I can see two advantages to the T4 approach in this article:

Performance

Consider the logging aspect for example. T4 will generate code at design time that accesses the source of the logging data (e.g. method parameters) directly. Therefore, performance is enhanced.

In the previous article's example, LoggingData extractors were executed every time a method is invoked to extract logging data via Reflection. With T4, most of this complexity is moved to design time, and therefore no extractors/generators need to execute at runtime.

Debugging Experience

With T4, we generate decorators at design time. So at runtime, when we debug, we will debug code that is specific to a certain interface/class, and therefore is simpler. This is true because generic code is always more complex than concrete/specific code.

Consider for example the logic that extracts logging data in the previous article's example. Such code does not exist in the T4 generated decorators so we don't have to debug it at runtime.

Please note however, that similar code (e.g. the LoggingDataCode generators) can be debugged at design time. For example, you can right-click on

the RetryAspectLocal.tt file in the Solution Explorer window and click Debug T4 Template. This allows us to debug the T4 template itself.

Conclusion:

T4 allows us to create templates that contain static code and code blocks that can generate code dynamically. This article provided examples on how to use T4 to create aspects: a retry aspect and a logging aspect. Using T4 to generate decorators at design time has some advantages over using Reflection at runtime. These advantages are better performance, and better runtime debugging experience ■

About the Author

Yacoub Massad

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.

Rahul Sahasrabuddhe

Server-less Architecture
- The Way Forward

In this article, we will discuss about serverless architecture. The cloud technology covers the whole gamut of IaaS to PaaS in terms of offerings. While the technology world evolves to scalable, flexible and modern cloud-based solutions; design & architecture of such applications is already in need of an overhaul. Serverless architecture paves the way for such an initiative. We'll explore this aspect more in this article.

Evolution of Apps & Architectures

Before we dive into the nuances of serverless architecture, let us first understand how the overall applications (or apps) landscape has evolved over the last few decades, and how it has had an impact on the architectures too.

Initially web only meant stateless, read-only web pages full of information. As the business requirements to put interactive data over the web grew rapidly, the stateless, read-only web pages got upgraded into web "applications". The applications added life to the Web, and made it more interesting, interactive and obviously more complex. This eventually resulted into applications being categorized as thick client apps (desktop applications) and thin client apps (web applications). The architectures for both these kinds of apps were still 3-tier or N-tier architectures, with server-client architecture as the base.

The client would mostly be a thick or a thin client application. The server would be the gateway for data management, business logic and workflows, wherein data would flow from applications to various systems embedded and integrated, using server side components. The architecture was very much dependent on the "server" for business data management. The server was indeed a core component of the architecture.

Then came in the mobile revolution. The mobile-enablement of consumers and corporate workforce, and the possibility of having internet on the mobile, gave another impetus to applications, which got termed as "apps". Apps is nothing but applications that are more modular, running on smaller & different form factors, and that perform specific tasks for an end user. Typical examples are expense management apps, ecommerce apps and of course, games. The app architecture is/was pretty much following the server-client model.

While the mobile revolution was happening, cloud-based offerings also started picking up. Initially the incentive for moving IT infra of an Enterprise, or even small business to the cloud, was to reduce costs. However, the rapid evolvement of cloud platforms like AWS and Azure added a new spin to the architecture. In addition to being able to host applications and reduce IT costs, cloud based offerings could also reduce costs further by decomposing the tightly coupled "server" component of a typical client-server architecture. This is now achieved by AaaS (i.e. Anything As A Service). So instead of just looking at Infra as a service (IaaS), now you can also look at a possibility of Backend as a Service (BaaS) or Function as a Service (FaaS).

And this is how a serverless architecture started making more sense.

So what is a Serverless Architecture anyway?

A serverless architecture, by definition, would mean that there is no server in the architecture. How is that possible, right? Traditionally, any architecture (2 tier or 3 tier generally speaking) would follow a typical server/client model. Irrespective of the type of application (thick client or thin client i.e. web), a client would be usually an interface to the end-users that would allow them to process some information, and the server would "serve" the requests coming from client - be it data or business logic etc. This is how it has been so far with respect to the classic traditional architecture.

A Shift in Thinking Process?

Over the years, there is clearly a shift in thinking in terms of how an architecture is envisioned. This is made more interesting by adding cloud-based scalable deployment to the mix. In traditional model, if you had to design a web application, you would think of UI, web services, business logic, database, and integration with third party systems. All of this would be sitting on the server. However, with AaaS, it is possible to further divide the server into more components that are reusable and consumable, and can also auto-scale up or down based on client needs. This actually is the definition of serverless architecture.

So a serverless architecture would imply all of the following:

1) An architecture that does not necessarily follow a classic & traditional client-server model where everything is sitting on the server (i.e. tightly bound by server boundaries and not really on a single machine).

2) An architecture that can take advantage of "elasticity" of the underlying hosted model of the cloud. This is a big plus.

3) An architecture where some or all the components on the server-side would depend on some 3rd party applications or services, that would manage some part of server side logic or business data or workflows. This is basically Backend as a Service (BaaS).

4) An architecture where parts of the server side logic may still be in control of the team that owns the development apps consuming the logic (Single Page Apps, Mobile apps or Web apps or anything else), however the logic would run in cloud as a self-contained service which would be consumed or invoked by these clients. This is basically Function as a Service (FaaS). AWS Lambda is a good example and we will learn more about it in the coming sections.

To sum it up, a **serverless architecture is basically a new design pattern that needs less effort in building your own server components**. Although it is not entirely serverless per se, it is probably server-less (i.e. less server dependency).

Enough of theory; let's look at an example

With the fancy definitions out of the way, let's see how this information is relevant for your applications. Let us understand this by taking an example of a typical online shopping application as seen in Figure 1.

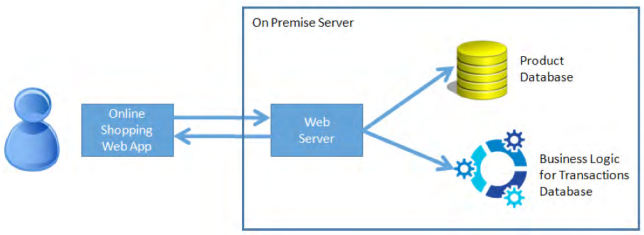


Figure 1: Shopping App Classic 3 Tier Architecture

In this online shopping app, a user is interacting with a web-based client for viewing and ordering certain items. The client could also be a mobile app (not shown here for the sake of simplicity). The user can first search through various items, and can choose specific ones for buying. The server will fetch relevant data from database and/or 3rd party systems, and then present product information to the user. The user would then be authenticated for ordering the items and an order will be placed. The application can also interact with any other third party components like payment gateway through a server to complete the transaction.

A few key points to note:

1) This is a classic 3-tier architecture hosted On-premise.

2) Server here in broader sense means all the server-side components including database, business logic etc.

3) The server is also responsible for calling any 3rd party components like payment gateway etc. Please note that the calls to such 3rd party components are routed through server and the control lies with the server only.

4) The client is mainly UX - most of the complex processing happens on the server-side - including authentication or payment. Please note that the server might be using some 3rd party services for this, but all of it is "hidden" from client.

5) This architecture design implies that the server is the entry-exit point for all interactions that are resulting from user interactions within the client itself. The server is also having all components tightly-bound within the boundaries of the infrastructure on premise.

Now let us look at how the same architecture would look like in the new serverless architecture world. It is obviously assumed that the online shopping application is on the cloud in the serverless architecture model.

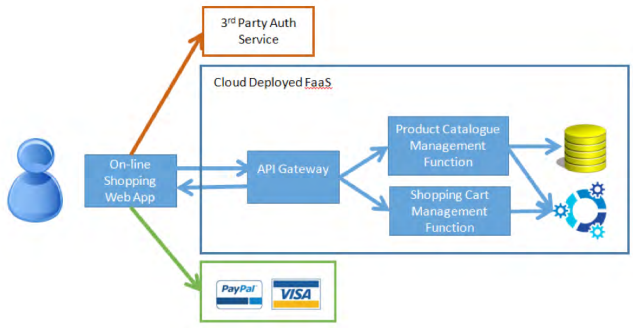


Figure 2: Cloud Deployed FaaS Architecture

In this architecture, the end user behavior and user-experience would not change at all. It is the client application that would now be consuming various Functions or micro-services. Some functions

or micro-services might be hosted on the cloud in a cluster, and would be developed by a specific organization that has created this online web application. On the other hand, some of these functions would be 3rd party functions that are directly consumed by the client app. Please note the use of word "Function" here. This new terminology is an indication in the shift of thinking in terms of how server or server-side logic is now envisioned in the serverless architecture.

In this example, only a few functions are being consumed directly & outside the bounds of server in order to keep it simple, and compare it with our earlier example. However the number of 3rd party Functions that an application can use or consume will depend upon the nature of app, and also the kind of relevant 3rd party functions that can be used.

Some key points to note here are:

1) The end user experience does not change in both kinds of architectural patterns.

2) The business logic is now organized as *Functions*, which makes it all the more modular. So from Infra as a Service (IaaS since apps are hosted on cloud), to now Function as a Service (FaaS) is the logical transformation that is happening in serverless architectures.

3) Some functions or modules like authentication, payment gateways etc. are sitting "outside" the logical definition or boundary of server. These are basically self-contained micro-services that are being consumed by the client application.

4) Because of all of these alterations in the server-side logic, the client has got more responsibilities now and thereby controls certain business logic directly (like authentication or payment gateway). So the client in traditional architecture and this one, are not the same anymore. This is an important implication of serverless architecture.

5) Your application could be using many servers (or services) but you will not be worrying about managing any of them at all. This is the beauty of serverless architecture.

Serverless Architecture - Hype or Reality?

Now that we have got a good grip on the *why and what* of a Serverless architecture, let us dive deeper to find out if this architecture makes sense for various kinds of apps out there, or is it just a hype that would die down slowly. For us to know this well, we need to understand as to *when and how* a serverless architecture would be relevant.

But before that, let us understand its pros & cons.

Serverless Architecture - The Pros & Cons

Let us look at the specific advantages & disadvantages of serverless architecture.

Following are some advantages:

1) **Lesser Total Cost of Ownership (TCO):** Since you are going to move out certain functions from your servers and consume the relevant services accordingly, the ownership of maintenance and scalability of those functions would lie with the service, or the function provider. This means that you will end up spending less money/ effort on the maintenance of this code. The total cost of ownership (TCO) would mainly include - maintenance, IT ops, development and testing, packaging & deployment costs.

2) **Elasticity of Operation:** Cloud based functions delivered as micro services offer an excellent ability to auto-scale up/down based on the work load. This is partly already fulfilled if your apps are on cloud. The same advantage is essentially now extended further.

3) **Technology Agnostic:** Servers are not bound to applications anymore, but they are rather serving

functions. This means that the functions could be developed in various technologies or languages, and yet can be consumed in applications hosted on other servers that are built on different technologies altogether.

4) **Reduced time to market:** This is more important or relevant for start-ups or organizations that want to launch the product in the market quickly - usually ahead of the competition. If you can reuse the functions that are provided as service, then it will reduce your overall calendar time for product launch.

5) **More Granular Billing:** Using FaaS also implies that it takes "pay as you use" methodology to a function call level.

Following are some disadvantages of limitations:

1) **Dependency on 3rd party:** If you are using a specific 3rd party or cloud provider Functions, then you are essentially locking yourself into a specific vendor - not at the infra level, but at the code level. This poses a serious risk to your product and it can be mitigated by choosing the right Function provider with good track record. Currently most cloud providers have FaaS, so this is mitigated up to a certain extent.

2) **Code Reliability:** Once you start using FaaS, then you are essentially dependent on the code written by others. So you need to be absolutely sure about how reliably the code is going to function even after periodic feature updates. This is very important when you are handling sensitive user information in your apps. You need to be very sure about the robustness and reliability of 3rd party apps in handling such data.

3) **Data Security:** When you end up using FaaS, your most precious data is going outside your control. For enterprises and specific ones like banks, this could be a serious cause of concern. However, this can be handled with having right checks and balances with Function providers, and also having appropriate SLAs.

4) **Simplicity of Operation:** Serverless might be about simplicity, but it does not always make your job simple. You have to take care of the plumbing code - session & state management, data flow between services etc. So it does not make things simple 'just like that'.

5) **Modular Design as Key Constraint:** The basic premise of having your application follow a Serverless architecture is that you need to have your code modular enough at a function level. It means that the code has to be clean & separable at a level of modules being self-sustaining, and self-contained code functions.

Guidance on Using Serverless Architecture

Let us be absolutely clear that your current on-going apps may not have to be serverless right away. You don't need to rush to the drawing board immediately, and start thinking of redesigning your app that has been working so well for you.

Here are certain key driving factors or scenarios that would indicate the use of serverless architecture.

1) You are migrating your existing app to the cloud, and it is not just IaaS play but more of an architectural revamp. Usually "move to cloud" initiatives are driven by infra cost-cutting & ROI calculations. So while doing so, if architectural revamp is not a priority, then serverless architecture at this point is not for you.

2) If you are going to deploy your application on cloud offerings like AWS, Azure, Google etc., then you can think of considering Serverless architecture based approach in your application design. However, you must know that this service is a very recent offering from most cloud providers (AWS Lambda is not more than 2-3 years old), so it means that the wide variety of things that you can achieve with such platforms, is limited to the extent of maturity of such offerings from cloud providers. Secondly, since these are pretty new and hence evolving, you may need to think through about the possibilities of being guinea pigs in the whole process. So tread water carefully here.

Cloud Based Serverless Offerings - What's on the menu?

Now let us look at specific offerings from leading cloud platforms that are based on serverless architecture concepts. As usual, some cloud platforms (viz. AWS) are ahead of the curve in terms of having an articulate offering for this, and some relatively newer cloud platforms (viz. Azure - no prizes for guessing that :)) are moving towards that goal.

Here is a short list of various platforms that are having serverless offerings.

- AWS Lambda
- Azure Functions
- Google Cloud Functions
- IBM Open Whisk
- Docker
- Iron.io
- Webtask

This link gives a detailed run-down on various offerings if you would like to build a Serverless application end to end.

For this article, we will focus on AWS and Azure cloud platforms.

AWS Lambda

AWS has two key services that are based on the serverless architecture model.

1) **AWS API Gateway:** a service offered for creating and managing APIs.

2) **AWS Lambda:** a service that allows you to run specific code functions. This is on the lines of FaaS model as described earlier. Using this service, we can create user-defined code functions, and they can be triggered from your application using HTTPS, similar to a typical service oriented architecture.

Usually both these services are used together to create a multi-tier application. The best part is that both these services can scale, based on the volume of requests made. They allow you to create a tier in your application that essentially takes care of various aspects of multi-tier application overheads like high availability, server/OS infra management, and so on.

Here are some key points to note:

1) You can consume Lambda functions from mobile/web apps etc.

2) They are essentially request-driven functions. Requests could be event-driven or data-driven.

3) They follow the implicit scaling model i.e. you don't have to worry about scaling up/down as it will happen based on workload.

4) Lambda functions can be invoked in the following ways:

a. Push Model: returns a response based on request made. Request could be event or a calling function.

b. Pull Model: happens when you subscribe to event source.

Here are some scenarios where you can use AWS Lambda (assuming you have your solution already deployed on AWS):

1) Document metadata processing or indexing

2) Image classification/indexing/processing

3) RSS feed content management

4) Media content validation/processing (audio/video)

5) Data cleansing/handling for imported/exported data

6) PDF watermarking

This is of course, not an exhaustive list. This list is to give you an idea about the kind of scenarios that can be considered for FaaS model. In addition to these scenarios, the following link provides more details about the reference architectures that can be used for AWS Lambda based solutions.

Azure Functions

Azure Functions is similar to AWS Lambda. Here is a quick summary of all the features:

1) Develop them in languages of your choice – C#, Node.js, Python etc.

2) You can use as pay-per-use and rely on Azure to auto-scale them based on workload.

3) Runtime for functions is open-source and available on GitHub

4) Full integration with development tools like GitHub, Visual Studio

5) It is ensured that the security aspect is taken care of through authentication using OAuth and other various providers.

6) Integration with various Azure features like DocumentDB, Azure Event Hubs, Azure Notifications Hubs & Azure Storage etc.
Following are the basic templates provided in Azure Functions:

- EventHub Trigger
- HTTP Trigger
- Blob Trigger, Queue Trigger
- ServiceBusQueue Trigger, ServiceBusTopic Trigger
- Timer Trigger
- Generic Webhook, Github Webhook

In terms of various scenarios to be used, here are some scenarios for which Azure Functions are being used effectively:

1) Processing files in Azure Blob with large file size by making a compressed copy of it for use/reference later for digital media requirements. Here is a step by step process explained in this link.

2) Analyzing the product information logs generated for an ecommerce site by pushing this data in SQL Azure or Excel for further analysis. This can be done by Azure functions in an automated mode with auto-scale based on workload changes.

There could be many more scenarios. In fact, the scenarios explained earlier for AWS Lambda can also be the right ones where Azure Functions can be used to a certain extent.

Conclusion:

We saw how Serverless architecture is picking up pace now, and how various cloud platforms are readying up their offerings to cater to these requirements. The key driver for going Serverless still continues to be the cost. We are essentially moving from cloudification of IT (IaaS) to Apps (PaaS) to now code (FaaS). These are certainly exciting times for overall app development space. And it is evident that like all other offerings Serverless Architecture will mature over a period of time and this has started in right direction.

The message here is clear – ***focus on code; not the servers*** ■

About the Author

rahul sahasrabuddhe

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.

A MAGAZINE FOR .NET AND JAVASCRIPT DEVS

- AGILE
- ASP.NET
- MVC, WEB API
- ANGULAR.JS
- NODE.JS
- AZURE
- VISUAL STUDIO
- .NET
- C#, WPF

We've got it all!

100K PLUS READERS

230 PLUS AWESOME ARTICLES

25 EDITIONS

FREE SUBSCRIPTION USING YOUR EMAIL

EVERY ISSUE DELIVERED

RIGHT TO YOUR INBOX

NO SPAM POLICY

SUBSCRIBE TODAY!

Daniel Jimenez Garcia

When it came to organizing your frontend code in ASP.NET MVC 5, you had several tools like HTML Helpers, Partial Views and Child Actions. ASP.NET Core introduces Tag Helpers and replaces Child Actions with View Components.

In this article, we will build a simple blogging site that integrates with Google accounts, and demonstrates how you can write readable, reusable and maintainable frontend code using these new features. I hope you enjoy it!

Organizing Frontend code in ASP.NET CORE a blogging example

Setting up a new Project with Google Authentication

Create the Project

Create a new *ASP.NET Core Web Application* project in Visual Studio 2015. Select the *Web Application* template, and finally select *Individual User Accounts* as the authentication method.

Once created, compile and launch the website and take note of your application url, for example http://localhost:64917. We will need this url later when configuring Google credentials. (You could also check the **launchSettings.json** file inside the Properties folder)

Now open a command line at the project's root folder, and execute the following command:

```
>dotnet ef database update
```

This command initializes the default database and applies the initial migrations, setting up all the tables required by the ASP.NET Core Identity framework.

Configuring the credentials

Before we even begin writing any code, we need to configure the credentials in the Google Developer Console. These are the credentials that our ASP.NET Core application will use when trying to get access to Google APIs.

To begin with, sign in with your google account in the Google Developer Console site https://console.developers.google.com. Then create a new project, give it a name and accept the terms:

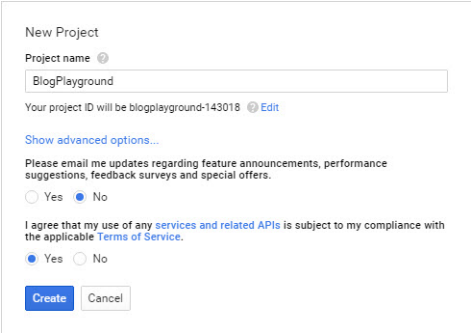


Figure 1: New project in the Google Developer Console

Once the project has been created, we need to enable the APIs that our web application will require. Navigate to the API Manager using the hamburger menu, then select the **Google+** link found within the Library of Google APIs, and finally click the enable button.

Next we need to setup the client credentials that will be used by our application to access these Google APIs. Go to the Credentials section using the left side menu. Click on the Create credentials button and select **OAuth client I**D:

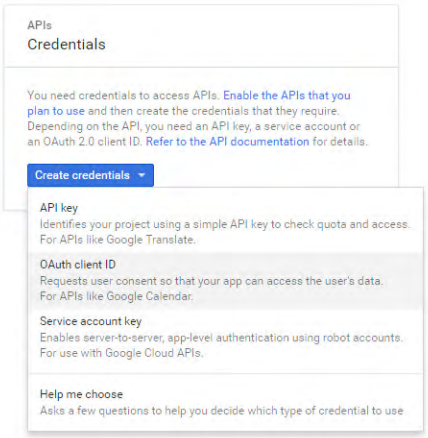


Figure 2: Beginning the creation of the OAuth credentials

Now configure the consent screen, the one shown to users asking them to grant our application access to their private data. At the bare minimum, you need to enter the name of your application.

With the consent screen configured, you can now select the *application type* of the credentials. Select the *Web Application* type and it will immediately ask you for the authorised origin urls and the redirect url:

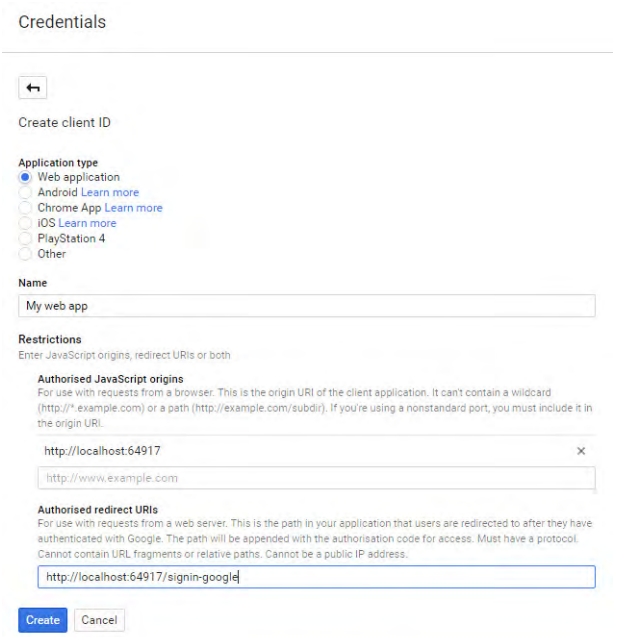


Figure 3: Configuring the credentials for our web application

Remember when I asked to take note of your application url? This is where you need it. Enter the base url as an authorised origin, for example http://localhost:64917. This is the origin from where these credentials will be used.

Now enter your base url followed by the path *signin-google* as the redirect uri, for example http://localhost:64917/signin-google. This is the url where Google will redirect users after they have logged in with their accounts. That particular url being the default callback of the Google Authentication package used in our ASP.NET Core application. (You could use a different redirect uri as long as you also configure it in your app Startup options)

Once you click Create, you will be presented with your Client ID and Client Secret. They can be viewed any time by clicking edit in your credentials:

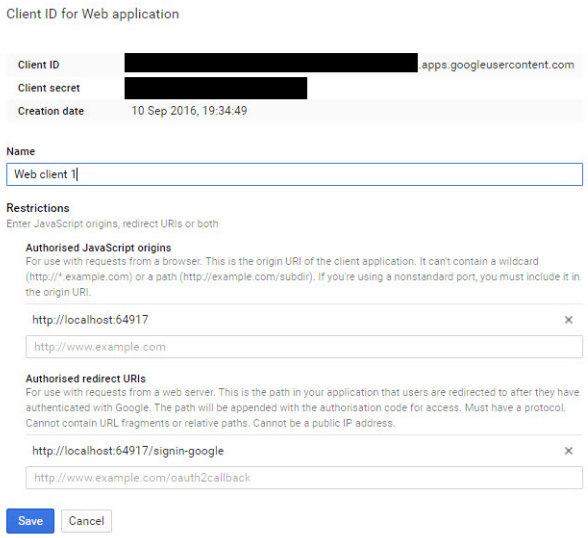


Figure 4: The Client ID and secret of the credentials

The Client ID and secret are quite important, as the Google Authentication middleware of our web application needs to use them.

This completes the setup in Google and we can go back to Visual Studio.

Enabling Google Authentication in the ASP.NET Core project

First let's add the middleware required for Google Authentication. In your **project.json**, add the following dependency:

```
"Microsoft.AspNetCore.Authentication.
Google": "1.0.0"
```

Make sure you save the file so Visual Studio restores the project dependencies, or manually runs `dotnet restore` in the command line.

Then update the **Configure** method of your **Startup** class, adding the Google Authentication middleware. The default project template contains a comment stating: *Add external authentication middleware* below. That's because the order in which the middleware is registered is critical, and this type of middleware should be registered **after UseIdentity and before UseMVC**.

```
app.UseGoogleAuthentication(new
GoogleOptions
{
  ClientId = "YourClientId",
  ClientSecret = "YourClientSecret",
  SaveTokens = true
});
```

You could just copy and paste your Client ID and secret. However I strongly suggest you to use the **Secret Manager tool** and to keep these secrets outside your source code, as explained here.

That's it, start your project and navigate to the Log in page. You will see a section for external providers with Google as the only option listed:

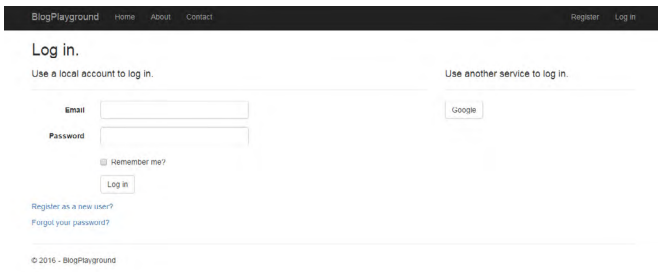


Figure 5, Log in now provides the option of using your Google account

If you click on that button, you will be taken to a page on Google's server where you can authenticate with your account and give consent to your web application (This is the consent screen configured earlier in the Google Developer Console):

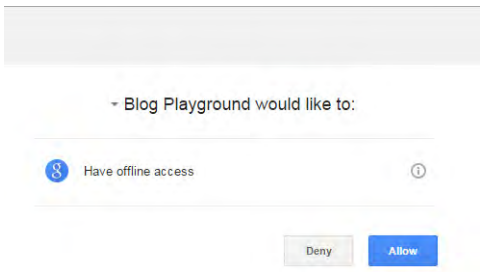


Figure 6, the Google consent screen

Once you click the Allow button, you will be sent back to the application where you will now be authenticated. During the first time, you also need to associate your Google account with a local account.

Storing the user profile picture and name

Our web application will retrieve the user name and profile picture from the user's Google account, and store them in the **ApplicationUser** class.

To begin with, extend ApplicationUser with these new properties:

```
public class ApplicationUser :
IdentityUser
{
  public string FullName { get; set; }
  public string PictureUrl { get; set; }
}
```

Then add a new migration and update the database by running these commands:

```
>dotnet ef migrations add user-profile
>dotnet ef database update
```

The user's full name is already available in the Identity Claims of the **ExternalLoginInfo** object after a successful Google authentication. (The Google middleware adds the email address and full name amongst others as the default scopes. You can request additional properties by adding them to the Scopes collection of the Google Options object, where the list of available scopes for the Google APIs is located at https://developers.google.com/identity/protocols/googlescopes)

We will however need to write some code to retrieve the profile picture. Remember the setting `SaveTokens=true` within the GoogleOptions in your Startup class? This will allow us to grab the access token and use it to retrieve a json with the basic profile information of the user.

This json contains a **picture** property with the url of the user's profile picture, and is retrieved from the following url: https://www.googleapis.com/oauth2/v2/userinfo?access_token=theAccessToken

Now that we know what to do, we can for example create a new service **GooglePictureLocator** with a method that takes an ExternalLoginInfo, and returns

the profile picture:

```
public async Task<string>
GetProfilePictureAsync(ExternalLoginInfo
info)
{
  var token = info.AuthenticationTokens
  .SingleOrDefault(t => t.Name ==
  "access_token");
  var apiRequestUri =
  new Uri("https://www.googleapis.com/
  oauth2/v2/userinfo?access_token=" +
  token.Value);
  using (var client = new HttpClient())
  {
    var stringResponse = await client.
    GetStringAsync(apiRequestUri);

    dynamic profile = JsonConvert.
    DeserializeObject(stringResponse);
    return profile.picture;
  }
}
```

Remember to register the service within the DI container in the Startup class. This way the code that needs to use the service just needs a constructor parameter of the service type, so an instance gets injected.

This utility service will be used when storing the user's full name and profile picture after:

- A new user successfully logs in with his Google account and associates it with a local account. This happens in the **ExternalLoginConfirmation** method of the **AccountController**.

- An existing user with a local account adds an external Google account. This happens in the **LinkLoginCallback** method of the **ManageController**.

In the ExternalLoginConfirmation, populate the FullName and PictureUrl properties before the user is created:

In the ExternalLoginConfirmation, populate the FullName and PictureUrl properties before the user is created:

```
var user = new ApplicationUser {
UserName = model.Email, Email = model.
Email };
user.FullName = info.Principal.
FindFirstValue(ClaimTypes.Name);

user.PictureUrl = await _pictureLocator.
GetProfilePictureAsync(info);
var result = await _userManager.
CreateAsync(user);
```

Do something similar in LinkLoginCallback, updating the user profile right before the redirect at the end of the method. (Save the updated properties to the db by calling `_userManager.UpdateAsync(user)`)

Make sure you register a new account once the changes are in place to retrieve the profile information from Google, so your account profile contains the retrieved information.

Show profile pictures in the navigation bar

Let's update the main navigation bar to display the user's name and picture:

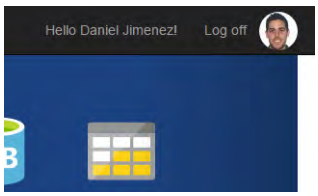


Figure 7: Updated navigation bar

While doing so, we will create a Tag Helper that renders the HTML of the profile picture. Tag Helpers let you encapsulate somewhat complex HTML rendering logic behind specific tags recognized by Razor that can be used from any of your views.

Create a Tag Helper for rendering profile pictures

Start by adding a new TagHelpers folder to your project, and a new class **ProfilePictureTagHelper** inheriting from the **TagHelper** abstract class.

Any class implementing **ITagHelper** (like any class that inherits from **TagHelper**) will be recognized as a Tag Helper, regardless of how they are named, and

where they are located. The default convention for matching tags with classes is based on the name, so ProfilePictureTagHelper matches the tag:

```
<profile-picture/>
```

When the view is rendered by Razor, the tag will be replaced with the result of the **Process** method of the **ProfilePictureTagHelper** class. As usual you can customize the convention; there are attributes for specifying the target tag element, required attributes and more.

The only additional requirement is to inform Razor about any assembly that contains Tag Helpers. Update **_ViewImports.cshtml** adding the following line (Where you add the assembly name, not the namespace!):

```
@addTagHelper *, BlogPlayground
```

Continue by adding the following code to the Tag Helper:

```
public ApplicationUser Profile { get;
set; }
public int? SizePx { get; set; }

public override void
Process(TagHelperContext context,
TagHelperOutput output)
{
  //Render nothing if there is no profile
  or profile doesn't have a picture url
  if (this.Profile == null || String.
  IsNullOrWhiteSpace(this.Profile.
  PictureUrl))
  {
    output.SuppressOutput();
    return;
  }

  //To be completed
}
```

This means our tag helper accepts 2 parameters and renders nothing when there is no profile or picture. (The later happening with users that use local accounts instead of google accounts).
We will complete it later. First, switch to **_LoginPartial.cshtml** and where the user is authenticated, retrieve its profile and use the profile picture tag helper:

```
@if (SignInManager.IsSignedIn(User))
{
  var userProfile = await UserManager.
  GetUserAsync(User);
  <form …>
    <ul class="nav navbar-nav pull-
    right">
      …
      <li>
        <profile-picture profile="@
        userProfile" size-px="40" />
      </li>
    </ul>
  </form>
}
```

We just need to complete the Process method of the Tag Helper so it renders a span with an inner <img/> element, like:

```
<span class="profile-picture">
  <img src="https://lh6.
  googleusercontent.com/...">
</span>
```

This is achieved with the following code:

```
output.TagName = "span";
output.TagMode = TagMode.
StartTagAndEndTag;
output.Attributes.SetAttribute("class",
"profile-picture");

var img = new TagBuilder("img");
img.Attributes.Add("src", this.
GetPictureUrl());
output.Content.SetHtmlContent(img);
```

- The GetPictureUrl function will not just provide the url. We need to control the size of the picture, so our site looks as expected. Luckily for us the Google picture accepts a query string like ?sz=40 that specifies the size in pixels:

```
private string GetPictureUrl()
{
  var imgUriBuilder = new
  UriBuilder(this.Profile.PictureUrl);
  if (this.SizePx.HasValue)
  {
    var query = QueryString.
    FromUriComponent(imgUriBuilder.
    Query);
    query = query.Add("sz", this.SizePx.
    Value.ToString());
    imgUriBuilder.Query = query.
```

```
    ToString();
  }
  return imgUriBuilder.Uri.ToString();
}
```

- You can also add the user's name as the title and alt attributes of the img element.

Finally, add some css rules to site.css so it looks similar to Figure 7:

```
.navbar .profile-picture {
    line-height: 50px;
}
.profile-picture img{
    vertical-align: middle;
    border-radius: 50%;
}
```

Provide a default image for local accounts

Right now if someone is using local accounts, the navigation bar will show no profile picture as we only have that information for users authenticating with Google accounts:

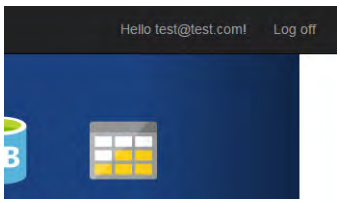


Figure 8: Local accounts have no profile picture

We can easily update the ProfilePictureTagHelper so it also renders a default placeholder for local accounts. Just google for a suitable default profile picture, and save it to the **wwwroot/images** folder.

The full url for this image can be generated using an **IUrlHelper** in the tag helper as Content("`~/images/placeholder.png`"), but we need to get access to an **IUrlHelper** instance first!

Add a constructor to the TagHelper which takes instances of `IUrlHelperFactory` and `IActionContextAccessor`.(You will also need to register `IActionContextAccessor` within the DI services in the Startup class)

Then create the following properties:

```
private bool IsDefaultPicture =>
this.Profile == null
|| String.IsNullOrWhiteSpace(this.
Profile.PictureUrl);
private IUrlHelper UrlHelper =>
  this.urlHelperFactory.
GetUrlHelper(this.actionAccessor.
ActionContext);
```

Update the GetPictureUrl method so it returns the placeholder when required:

```
if (this.IsDefaultPicture)
{
  return this.UrlHelper.Content("~/
  images/placeholder.png");
}
```

Then update the **Process** method so it doesn't suppress the output anymore when there is no PictureUrl. Instead, let it proceed and just update the code so it forces the width and height in pixels for the default profile picture. (This is the simplest solution to ensure the single default placeholder picture has the dimensions required)

```
if (this.IsDefaultPicture && this.
SizePx.HasValue) {
  img.Attributes.Add("style",
  $"height:{this.SizePx.Value}
  px;width:{this.SizePx.Value}px");
}
```

That's it, now local accounts will show the default placeholder:

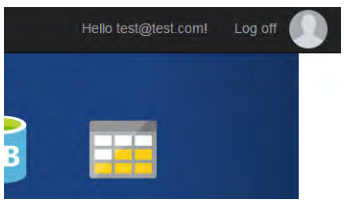


Figure 9: Placeholder profile picture for local accounts

Building the blogging site

The objective of the article is building a simple blogging site, so we need pages where users create articles for display. Let's just use scaffolding to quickly generate the required controller and views.

I will stay focused on the frontend code, if you have any issues with the rest of the code please check the article code in GitHub.

Create new Articles controller and views

Create a new Article class inside the Models folder:

```
public class Article
{
  public int ArticleId { get; set; }
  [Required]
  public string Title { get; set; }
  [Required]
  public string Abstract { get; set; }
  [Required]
  public string Contents { get; set; }
  public DateTime CreatedDate { get;
  set; }

  public string AuthorId { get; set; }
  public virtual ApplicationUser Author
  { get; set; }
}
```

This is a POCO model that Entity Framework will be able to store and whose relationship with the ApplicationUser model will be automatically managed. (Read more about relationships here.)

Since it has a relationship with the ApplicationUser model, it needs to be added as a new DbSet in the default **ApplicationDbContext** class:

```
public DbSet<Article> Article { get;
set; }
```

At this point, you probably want to add a new migration, go ahead and do so. Once you are ready, we will scaffold everything we need. (If you are already using the secret manager for storing your Google ClientId and Secret, check this issue in github)

Right click the Controllers folder and select Add > Controller. Then select *MVC Controller with views, using Entity Framework*. Select the Article class as the model and the existing ApplicationDbContext as your data context, so Articles and ApplicationUsers live in the same context.

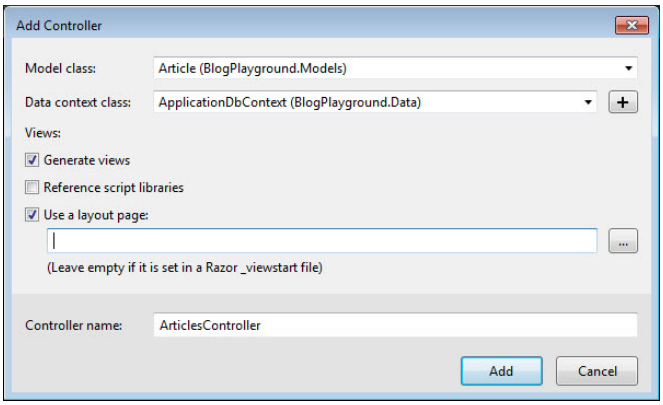


Figure 10, scaffolding the articles

Once scaffolding is completed, let's apply some small changes:

- I will ignore the edit view, feel free to remove it along with corresponding controller actions.

- Add the [Authorize] attribute to the Create and Delete actions, so only logged in users can add/remove them.

- Leave just the Title, Abstract and Content in the Create view. Use the textarea tag helper for both the Abstract and Content, for example:

```
<textarea asp-for="Abstract"
class="form-control" rows="5"></
textarea>
```

- Set the AuthorId and CreatedDate in the Create method of the controller. (You will need to inject an UserManager<ApplicationUser> in the controller constructor)

```
article.AuthorId = _userManager.
GetUserId(this.User);
article.CreatedDate = DateTime.Now;
```

- You will see that scaffolding already created a query for the Index page that includes the Author information within a single query. However you will manually need to do the same in the Details action so it also loads the Author within the same single database query:

```
var article = await _context.Article
  .Include(a => a.Author)
  .SingleOrDefaultAsync(m =>
  m.ArticleId == id);
```

You should now have a functional site where users can write articles which are then displayed inside the index view.

Polish the views using partials and tag helpers

We will update the article views so they show the author details and look nicer (within the boundaries of my limited design skills) while its code stays clean. As Bootstrap is included by default, I will base my design on it.

Let's start with the Index view. Change it so we now have a title followed by a container div. Inside the container, create a main left side column and a smaller right side column.

- The right side contains a link for creating a new article, styled as a main button.

- The left main side is where articles will be rendered. Use a list with an item for every article. Each item will be also divided on left and side columns.

- Display the author profile picture (using our tag helper), name and creation date on the smaller left side column.

- Finally display the title and abstract on the right side column. Don't forget to render the title as a link for the details page of that article.

Once finished with the html and css changes, it should look similar to the following screenshot:

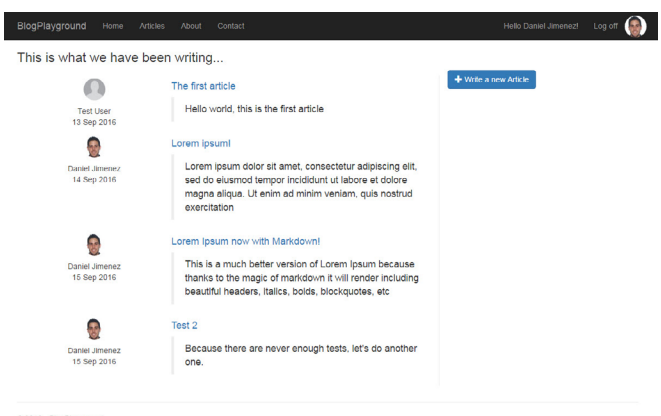


Figure 11: Updated articles index view

Continue by extracting the razor code rendering each article into an old fashioned partial view. We will later reuse it on other views, keeping their code clean and simple. This partial view should be close to mine, depending on how closely you have followed my design:

```
@model BlogPlayground.Models.Article
<div class="row article-summary">
 <div class="col-md-4">
  <profile-picture profile="@Model.Author"
  size-px="40" />
  <p class="text-center author-name">
   @(Model.Author.FullName ?? Model.
   Author.UserName)
  </p>
  <p class="text-center">
   @Model.CreatedDate.ToString("dd MMM
   yyyy")
  </p>
 </div>
 <div class="col-md-8">
  <h4>
   <a asp-action="Details" asp-route-
   id="@Model.ArticleId">
   @Model.Title
   </a>
  </h4>
  <blockquote>
  <p>@Model.Abstract</p>
 </blockquote>
 </div>
</div>
```

With the partial created, the Index view should now be quite simple and clean:

```
@model IEnumerable<BlogPlayground.
Models.Article>
<h3>This is what we have been
writing...</h3>
<div class="container">
 <ul class="col-md-8 list-unstyled
 article-list">
 @foreach (var article in Model)
 {
  <li>
   @Html.Partial("_ArticleSummary",
   article)
  </li>
 }
 </ul>

 <div class="col-md-4">
  @*create new article button*@
 </div>
</div>
```

Now it's the turn of the Details view. Follow the same approach with a container divided into a main left side showing the article details, and a smaller right side showing the links.

- In the main left side, render the _ArticleSummary partial followed by the article contents.

- In the right side, use a Bootstrap button group and provide links to write a new article, delete the current one or go back to the list.

```
@model BlogPlayground.Models.Article
<div class="article-page container">
 <div class="col-md-8 article-details">
  @Html.Partial("_ArticleSummary",
  Model)
             <hr />
 <p>@Model.Contents</p>
</div>
<div class="col-md-4">
  @* button group *@
 </div>
</div>
```

Once finished, the details page will look similar to this screenshot:

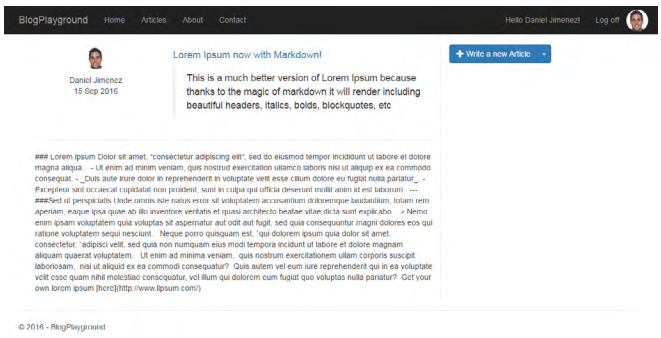


Figure 12: Updated article details page

As you can see, partials are still useful for keeping views clean and reusing code!

Render markdown contents

Rendering the article contents as an unformatted block of text in a blogging site is not very useful. An easy way of improving this situation would be using markdown.

- Markdown is a simple markup language that can be included in plain text and is later formatted into HTML by tools. Stack Overflow and the readme.md files in git repos are just 2 very common examples where it is used.

The article contents will continue to be entered and stored as plain text, which can contain markup syntax. We will then create a new Tag Helper that will render any string as the markdown processed HTML.

Add one of the several nuget packages that can convert a string with markdown syntax into HTML, for example:

```
"Markdown": "2.2.0",
```

Add a new **MarkdownTagHelper** class to the TagHelpers folder. This one will be quite simple; it will accept a source string and render the markdown processed version of that string:

```
public class MarkdownTagHelper :
TagHelper
{
 public string Source { get; set; }
 public override void
 Process(TagHelperContext context,
 TagHelperOutput output)
 {
  output.TagName = "div";
  output.TagMode = TagMode.
  StartTagAndEndTag;
  output.Attributes.
  SetAttribute("class", "markdown-
  contents");
  var markdown = new Markdown();
  output.Content.
  SetHtmlContent(markdown.
  Transform(this.Source));
 }
}
```

Notice the usage of **Content.SetHtmlContent** when adding the processed string to the output. This is required as otherwise the HTML tags in the transformed string would be HTML escaped!

Once the Tag Helper is ready, use it in the Details view for rendering the article contents:

```
<div class="col-md-8 article-details">
```

```
 @Html.Partial("_ArticleSummary", Model)
 <hr />
 <markdown source="@Model.Contents">
 </markdown>
</div>
```

Now the Details page looks way more interesting than it did:

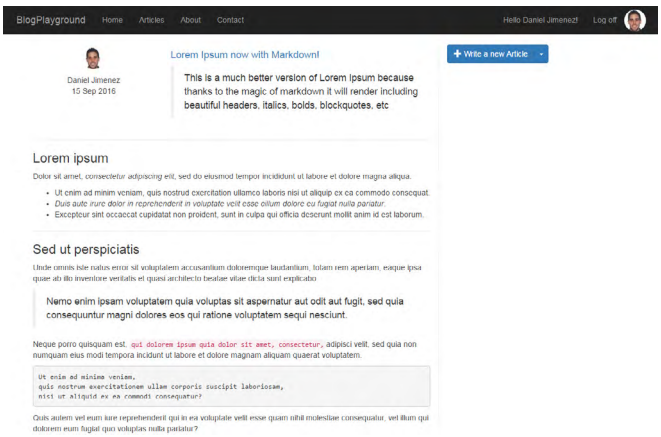


Figure 13, article details with processed markdown

Adding a Latest Articles View Component

In ASP.NET Core, View Components have replaced Child Actions as the way of encapsulating components into reusable pieces of code.

- The component logic won't be part of a controller anymore; instead it will be placed inside the View Component class. The views are also placed in component specific folders. This encapsulates the logic and view in a better way than the Child Action approach with partial views and controller actions.

- View Components are also lighter. They are invoked directly and parameters are directly passed into them, without involving the same pipeline than controller actions. (Like filters or model binding)

- They are not accessible by default over HTML. (If you need to, it's possible to write a controller action that returns a View Component)

- They support async code, which was not possible with Child Actions.

A View Component is the right approach for building a reusable widget that renders the latest articles in our blogging site. As you will see, it will also be straightforward to build on top of our current code.

Add a new ViewComponents folder to the project and create a new class **LatestArticlesViewComponent**.

- Any class deriving from ViewComponent, decorated with [ViewComponent] or whose name ends in *ViewComponent* will be recognized by ASP.NET Core as a view component.

Just inherit from the ViewComponent abstract class and implement the **InvokeAsync** method. This method will receive the number of articles to retrieve, fetch them from the database, and finally render them:

```
public class
LatestArticlesViewComponent:
ViewComponent
{
 private readonly ApplicationDbContext
 _context;
 public LastArticlesViewComponent
 (ApplicationDbContext context)
 {
  _context = context;
 }
 public async Task<IViewComponentResult>
 InvokeAsync(int howMany = 3)
 {

  var articles = await _context.Article
  .OrderByDescending(a => a.CreatedDate)
  .Take(howMany)
  .ToListAsync();
  return View(articles);
 }
}
```

The default convention expects the view located in the folder **Views\Shared\Components\LatestArticles.** Since we are not specifying any view name, the convention is that the view must be named **Default.cshtml**.

The razor code inside that view will simply render a list with the latest articles. Again we use the **_ArticleSummary** partial in order to render each item:

```
@model BlogPlayground.Models.Article
<div class="last-articles">

  <h4>Latest articles:</h4>
    <ul class="list-unstyled last-
    articles-list">
    @foreach (var article in Model)
    {
      <li>
        @Html.Partial("_ArticleSummary",
        article)
      </li>
    }
  </ul>
</div>
```

Once the View Component is finished, we can include it in the sidebar of both the Index and Details views. For example in the Index view, right below the Write a new Article button:

```
<div class="row">
  @await Component.
  InvokeAsync("LatestArticles", new {
  howMany = 2 })
</div>
```

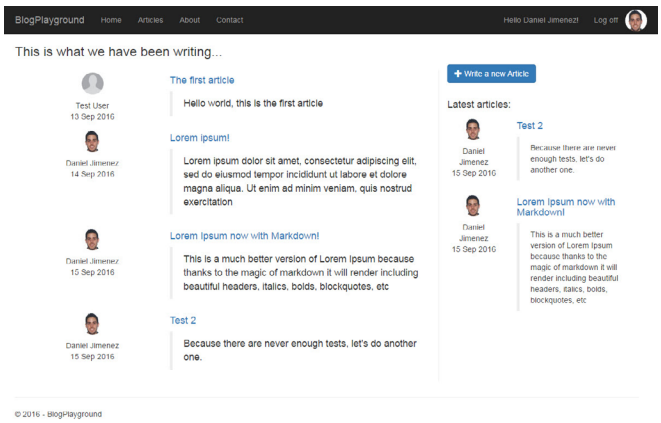


Figure 14: Index page including latest articles widget

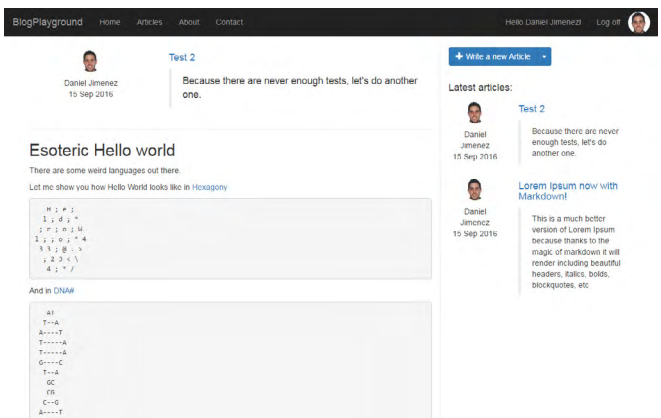


Figure 15: Details page including latest articles widget

Conclusion

We have seen how we can build non-trivial sites while keeping our frontend code clean, maintainable and reusable. Previous techniques like partial views are still relevant, but the addition of Tag Helpers and View Components in ASP.NET Core makes it even easier to achieve those objectives.

- Tag Helpers are great for encapsulating blocks of HTML code that you might need in several places and for rendering concerns like *img* attributes or markdown strings. The profile picture and markdown Tag Helpers are good examples.

- View Components let you encapsulate together business logic and views in a powerful and simple way. The latest articles widget is the perfect example of what can be easily achieved.

The site built in this article wraps everything into a simple but functional blogging site. Although interesting, bear in mind this is only meant as an example. Due to time and space constraints, I have happily overlooked many issues you would face in a real project like searching, pagination, different social providers like Facebook or Twitter, profile management and many more! ■

Download the entire source code from GitHub at bit.ly/dncm27-aspnetcore-blogging

About the Author

daniel garcia

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.

Vikram Pendse

Azure Elastic Pools for SaaS Applications

In enterprises, especially Product based ones, "SaaS" (Software as a Service) is a frequently used term where product owners sell their product to multiple customers to enhance their business. In theory, it sounds very simple, but SaaSification is full of challenges. Besides scalability and performance, cost and resource consumption also are key aspects. As far as Databases are concerned, services like "Elastic Pool" comes in handy to calculate and minimize the challenges that may arise. In this article, we will explore Elastic Pool in Azure with a case study, and see how it helps to scale and achieve economical resource consumption especially in SaaS scenarios.

Elastic Pool for Azure hosted SaaS applications

Current Databases scenario and options available in Azure

Before we start discussing about Azure Elastic Pool, let's discuss the various Database options available in Azure. For enterprise applications there are a lot of open and NoSQL database offerings available in Azure like MongoDB, DocumentDB etc. There are two main flavors of SQL in Azure - Azure SQL (PaaS) and SQL Server (IaaS). Both have good set of features.

IaaS based solutions are generally used in enterprises if there is a requirement of SSIS, SSRS and SSAS, since PaaS based SQL Azure does not provide or support these services. SQL database is used in both scenarios of Predictable and Unpredictable workloads. When we talk about SaaS, we almost always have unpredictable workloads. In this case, it becomes complex to architect the storage/database distribution and pattern, and maintenance as well. Following are the challenges

we normally see in SaaS applications, as far as the Database is concerned:

1. Customers who need their unique database instance can grow in an unpredictable manner

2. Customers are scattered geographically

3. Customers may have unique demands of performance (CPU, Memory and IO)

Once enterprises start tackling these issues, most of the times they end up in either over provisioning of resources, or under provisioning them. With this, they neither meet performance expectations, nor they save anything on the costs. Hence Elastic Pool becomes a natural choice to overcome these barriers.

What is Elastic Pool?

Elastic Pool is a service offered by Azure which helps you to run multiple independent and isolated databases, and can be auto scaled across a dedicated tier of private resources pool. It helps you to set a policy over a group of elastic databases, and helps to manage performance individually, instead of enterprises managing them exclusively. While this is a great feature and service, not all databases are eligible/candidate for an elastic pool. Databases with variable activities over time, and where all databases are not active at the same time are eligible for Elastic pool. In that case, they can share resources i.e. eDTUs (elastic Database Transaction Units).

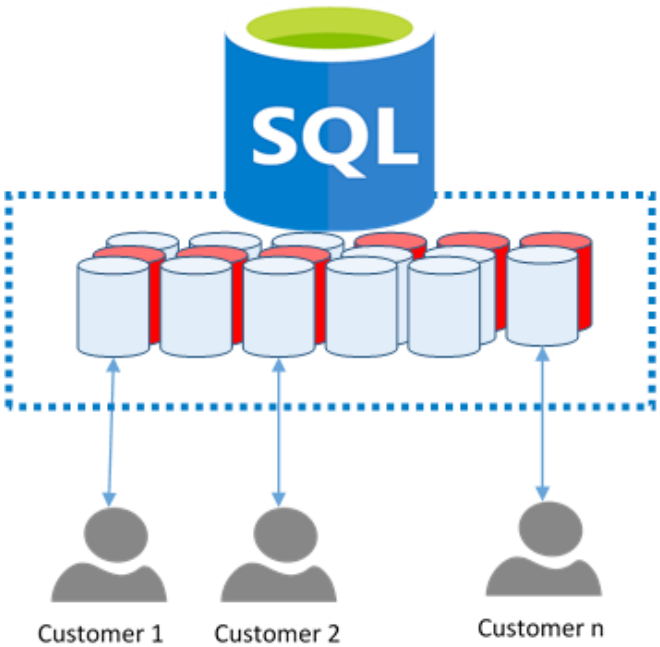
There are different tiers like Basic, Standard and Premium, and each of the pools are given some predefined eDTUs. Within the pool, databases can scale on demand based on the scaling parameters, and as per the load. So the entire price or cost model is based on the consumption in eDTU (which is a relative measuring unit for SQL databases to handle resource demands). Hence instead of provisioning the resources for a particular database, you can do it over the pool, and thus management of all those databases also becomes easier.

Case Study

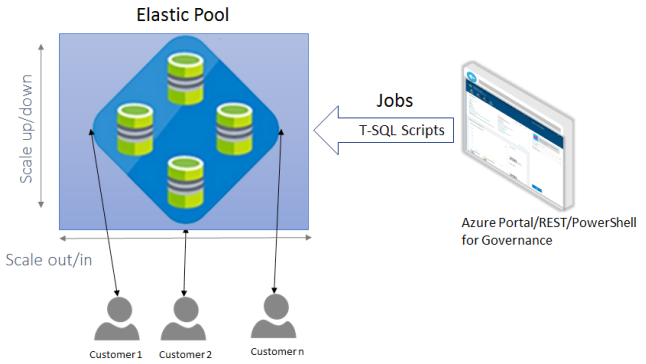
A large enterprise say A2ZDNC (fictitious name) has a SaaS solution with multiple SQL Azure database instances on a single SQL Azure Server. A2ZDNC has multiple customers, and each one of these customers have different demands for performance. A2ZDNC also has its own instance of the database. Master Database keeps metadata of all customer databases, as well as its distribution and configuration. A2ZDNC recently found that to meet customer demands of scaling and performance, they always end up over provisioning, which is in turn impacting performance and cost.

Let's see how Elastic Pool can solve this problem that A2ZDNC is facing.

Here's a diagrammatic representation of the current scenario representing instances based on customer demands of scaling and performance. The Databases in Red shows over provisioning.



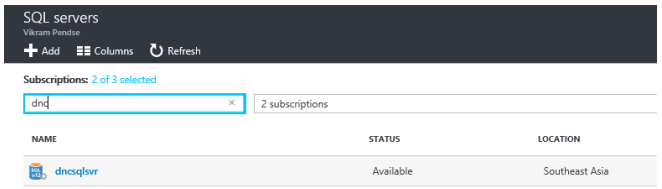
With this current architecture, Cost and Performance are not balanced. Now with Elastic Pool, this is how the Architecture will look like:



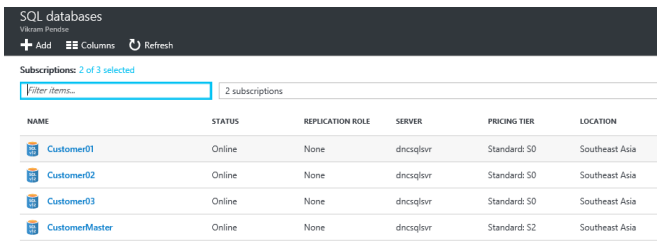
We will understand this architecture and its advantages in the forthcoming sections.

Azure SQL Server Databases

We have multiple databases within the server blade which are created for individual customers as per need, and which is one of the fundamental requirements of SaaS. Select SQL Server which is created and hosting multiple databases.

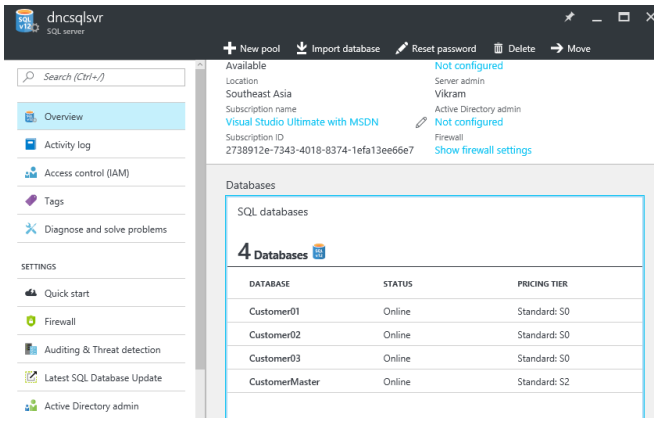


You can see that each customer has their own instance of database as per their unique requirement. There is a Master database which is acting here as a transactional database and has all the relevant metadata of all the databases created for individual SaaS customers.

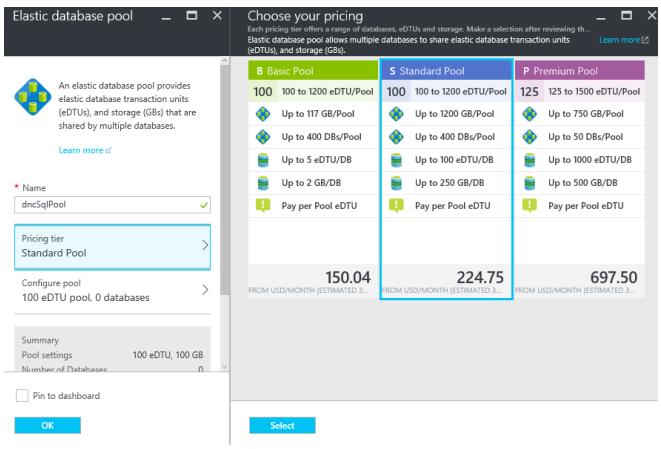


Creating Elastic Pool and adding databases to the Pool

To create a new Elastic Pool, you need to first start SQL server blade where you have all the databases. On the top menu, you will see a "New Pool" option.

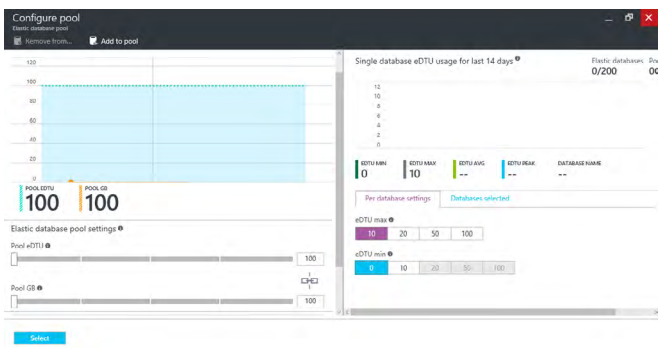


You need to click on that and then you will get a new blade where you need to fill mandatory parameters and information.



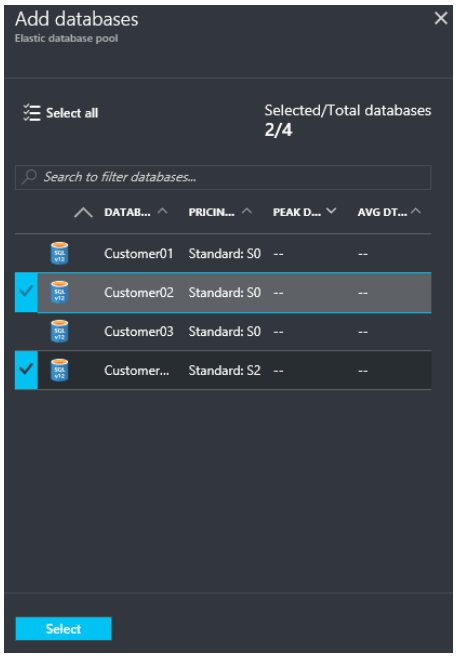
Give a meaningful name to your pool. Then choose the Pricing Tier. There are three pricing tiers available as Basic, Standard and Premium which enterprises can choose based on their requirements. Currently we are choosing the Standard Pool. If you compare the tiers, the primary classifications are based on Size, Number of Databases and eDTUs per database supported.

Now configure the pool by setting eDTU Max and Min per database instance, and overall Pool eDTU and size in GB as shown here.

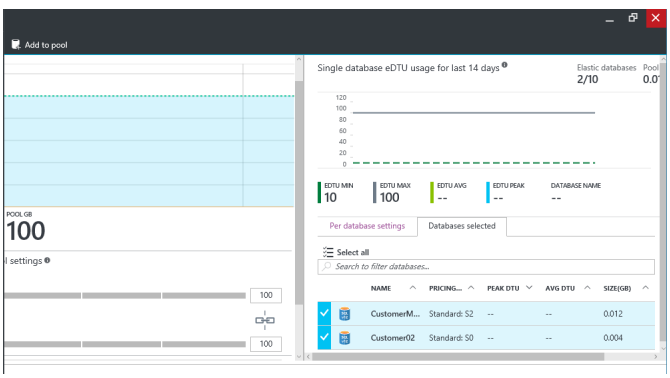


Which Databases go in the Elastic Pool?

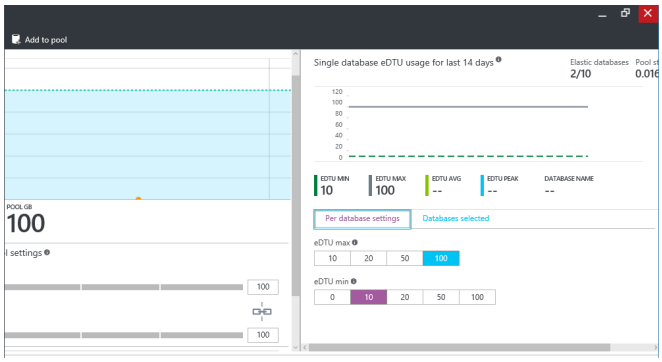
By nature, elastic pools work well for a lot of database which are not active at the same time. Here Azure uses Machine Learning (ML) and gives you recommendations as well. For example, if there is a CRM service which is talking to one of the database instances, usually not all customers would be running queries at same time on the database. Say on an average, these instances are active up to only 5% time during the day. So such databases are ideal candidates to go in the pool. This way Azure will provide us recommendations based on the historic usage. Alternatively we can always add these databases manually.

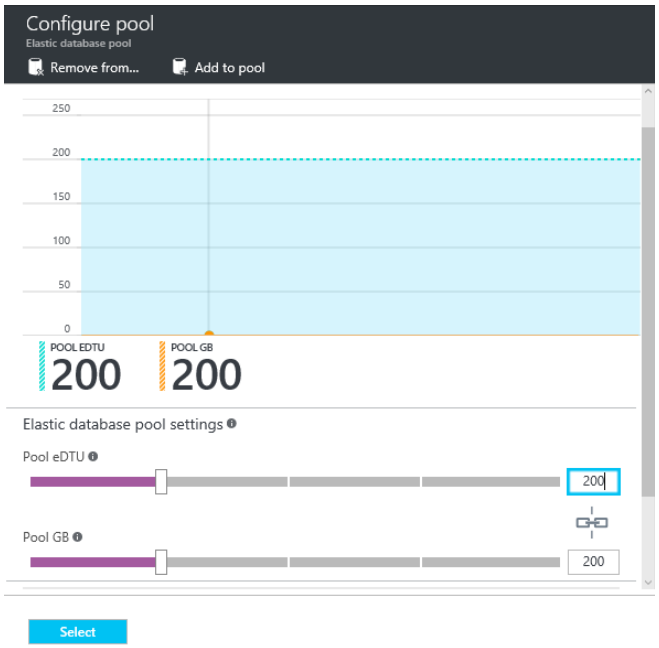


Note that in Azure you can have multiple databases in one pool, and you can have multiple such elastic pools as well. Everything depends on the nature of your SaaS application, and Database demand and usage.



Now you can configure eDTU Max and Min along with Pool eDTU. Note that you can add additional databases to the pool, as well as remove databases from the existing pool as well.



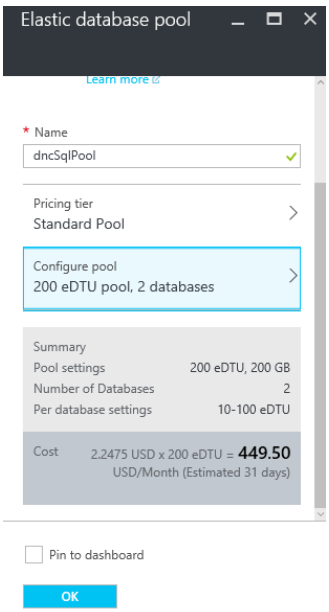


So the choice of databases to be added to the elastic pool is driven by various other important factors like shared eDTUs between multiple databases.

Elastic Pool configured, what will be my cost after pool creation?

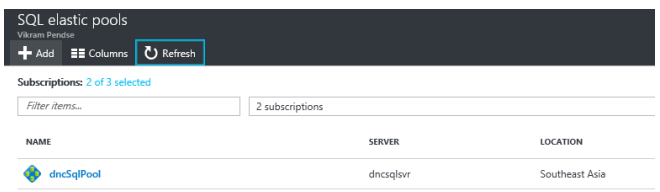
In the existing Pool blade, on the Summary tab, you can see all the configuration details along with the probable price/consumption for the month (31 days). This now gives you a clear vision on the cost and savings you are going to get, post the pool goes active. Based on the eDTU, you can always change configurations. Note that SQL also helps to evaluate

the performance and usage of the databases based on their historic usage.



Monitoring Pool

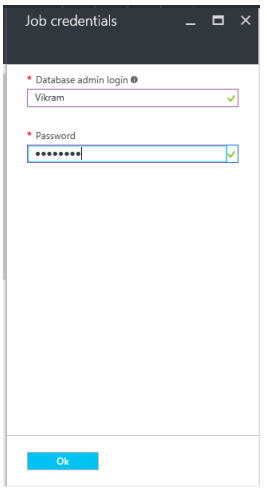
Once your pool is active, you can access it from the Elastic Pool blade. The Overview tab gives you a wide range of monitoring tools, and you can customize the monitoring charts as well. It also allows you to add Alerts on a pool, and you can get notifications related to it as well.



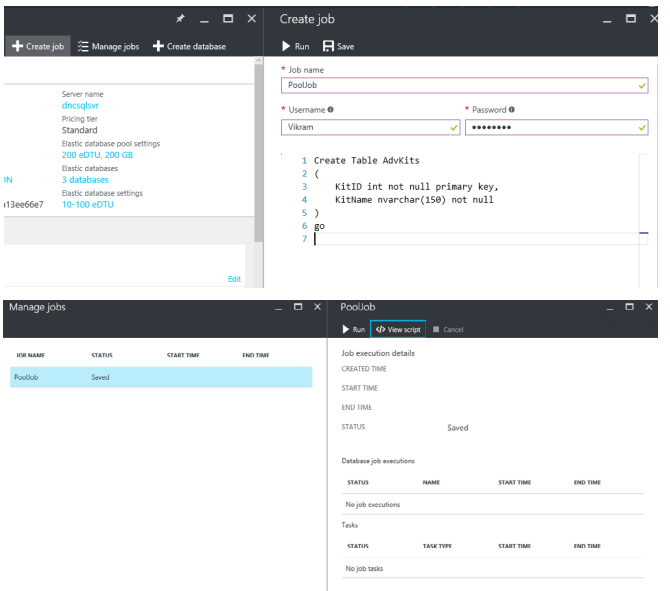
You can create multiple pools depending upon your SaaS design and architecture.

Elastic Pool Jobs

Managing the Pool is one of the key activities in Elastic Pool, and it is done using Jobs. You can quickly create any job from the Elastic Pool blade by choosing “Create Job” and “Manage Jobs” options. Note that the jobs have to be created using your Administrative credentials.



The Portal will ask you to key in the required Administrator credentials to create and run jobs. Jobs also deliver all the failure reports, and they can be run in an auto-retry mode as well. Jobs bring in a lot of improvements and automation, as far as managing the pool and databases in it is concerned.



The advantage of Jobs is you can easily manage large number of SQL instances easily from script/T-SQL to all the databases in the Pool. You can also apply DACPAC with the help of Jobs. DACPAC enables data-tier developers and database administrators to package SQL Server objects into a portable artifact. This can also be done using PowerShell as well. Jobs can consist of performance queries, schema changes, collect data from multiple databases into a single master database for processing and reporting.

Case Study Conclusion:

With the old architecture, A2ZDNC was spending more due to over provisioning, and the architecture also had a negative impact on performance. With Elastic Pool solution in Azure, A2ZDNC now can easily scale and make optimal usage of database resources, and thus can see a good amount of cost savings in their Azure billing. Moreover, they also get the flexibility to apply changes and do administrative operations to multiple databases at the same time. With built-in automatic retries in case of transient failures of elastic jobs, A2ZDNC now has full control on the databases. This helps to bring down the cost due to overprovisioning, and also boosts performance.

Conclusion:

Elastic Pool or Pool of SaaS elastic databases share a set of resources (eDTUs) for an industry best price/performance ratio. Individual DBs in the pool auto scale within set resource parameters. Single database per tenant model provides full tenant isolation and security. Intelligently auto-managed performance and predictable budget for the pool. Tools for simplified management tasks across 1000s databases & pools. All DBs use the same pool when they are active. With this benefits from Elastic Pool, your SaaS solutions thus becomes cost effective and highly effective ■

About the Author

vikram pendse

Microsoft® MVP Most Valuable Professional

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

Benjamin Jakobus

Bootstrap has become the world's favourite framework for building responsive web-projects. With the Bootstrap 4 Beta release just around the corner, it is time to take a more detailed look at what the project has to offer, what has changed and what one can expect when migrating over from Bootstrap 3. As such, this article will cover some of Bootstrap 4's major differences to its predecessor.

Exploring BOOTSTRAP 4

> *Bootstrap, originally named "Twitter Blueprint" was developed by Mark Otto and Jacob Thornton at Twitter.*

Bootstrap 4 has been in the making for over 2 years, and has yet some way to go until it is stable. However, as we are nearing a beta release, we can confidently list some of the major features and changes to expect when migrating from Bootstrap 3. To focus in on the latter; a migration will not be trivial, as Bootstrap 4 is a complete re-write of its predecessor.
Unlike Bootstrap 3, Bootstrap 4 is written in Sass (Syntactically Awesome Stylesheets). This is bad news for those sites that rely heavily on Bootstrap 3 variables.

The second major change comes with the components that Bootstrap 4 supports - or, in fact, whose support it drops: as of Alpha 4, Bootstrap no longer supports panels, wells or thumbnails. Instead, these have been replaced with one single concept: *cards*. Cards aim to serve the same purpose than wells and panels, but are less restrictive by supporting different types of contents, such as lists, images, headers and footers. The badge component has also been dropped: instead, Bootstrap requires you to use the label component.

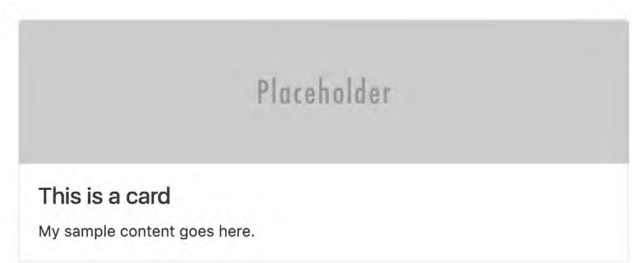


Figure 1: An example of Bootstrap's the card component: the top area of the card is taken up by a placeholder image, whilst the bottom area displays the card's content.

Reboot: The new Normalize

In order to provide a foundation from which all HTML elements would have a consistent appearance across browsers, Bootstrap 3 used Normalize.css. Bootstrap 4 on the other hand uses an improved version of Normalize.css called Reboot. As per the Bootstrap 4 documentation: "*Reboot builds upon Normalize, providing many HTML elements with somewhat opinionated styles using only element selectors.*" (https://v4-alpha.getbootstrap.com/content/reboot/). Although an important change, the migration from Normalize to Reboot goes largely un-noticed when actually using Bootstrap during development.

The Grid System

Just as with Bootstrap 3, at the core of Bootstrap 4, lies its grid system: a collection of CSS classes and media queries that allow developers to produce a responsive page layout. With Bootstrap 4, the grid system itself has been completely over-hauled, however luckily, most changes themselves are not breaking (see section "Notable classname changes"), as most class names have stayed the same - although it does introduce a new grid tier with a breakpoint at 480px, and a new `sm` grid tier. This gives Bootstrap 4, four grid tiers, instead of 3: `xs`, `sm`, `md`, and `lg`.

The addition of the `sm` grid tier means that Bootstrap now has five breakpoints:

- An extra-small breakpoint for handheld devices that boast a smaller screen than normal (0px)
- A small breakpoint aimed at phones / handhelds (544px)
- A breakpoint for screens of medium size, such as tablets (768px)
- A breakpoint for large screens - that is, desktops (992px)
- An extra-large breakpoint to support wide-screen (1200px)

These five breakpoints are defined as properties of $grid-breakpoints, and are defined in _variables.scss:

```
$grid-breakpoints: (
  // Extra small screen / phone
  xs: 0,
  // Small screen / phone
  sm: 544px,
  // Medium screen / tablet
  md: 768px,
```

```
// Large screen / desktop
lg: 992px,
// Extra large screen / wide desktop
xl: 1200px
) !default;
```

Furthermore, all grid classes can be disabled by setting the `$enable-grid-classes` variable inside _variables.scss to false, and recompiling Bootstrap. It should be noted that Bootstrap 4 places a great emphasis on customisation, and as such, _variables.scss offers any easy way to configure and customize Bootstrap.

Flexbox

By setting `$enable-flex` inside _variables.scss, Bootstrap 4 allows you to enable Flexbox support for the grid system, media components and input groups. To readers unfamiliar with Flexbox: Flexbox is a layout model that allows for the building of layouts that are more complex than the usual CSS2 layouts such as `block`, `positioned` or `table`. Whilst not aimed at replacing existing page layout functionality, the layout model is ideally used when needing to order or scale elements or needing more advanced vertical or horizontal alignment capabilities.

Typographic units of measurement

Typographic units of measurements are used to define the size of a page's font and elements. Bootstrap 4 uses a different type of typographic measurement to its predecessor, moving away from using pixels (px) to using root em (rem). The use of pixels has, in the past, resulted in rendering issues when users change the size of the browser's base font. Since rem is relative to the page's root element, sites built using Bootstrap 4 will scale relative to the browser's base font. Furthermore, the global font-size increased by 2 pixels, from 14px to 16px.

Style and appearance

Aside from font-size changes, the overall look and feel of Bootstrap has not drastically changed. There are some minor changes around the size of elements, padding and margins, as well as the primary context colour, whose background colour has changed from #2f79b9 to #0072db. Bootstrap 4 also no longer ships with Glyphicons, making the framework more lightweight but also requiring you to either manually integrate them, or choosing an alternative.

Improved form controls

Possibly one of the most exciting new features that ship with Bootstrap 4 are its improved form control styles. The current Alpha release comes with validation styles that finally render the many third-party Bootstrap validation plugins obsolete (see figure 1), and as such greatly improve the "out of the box" experience. And on top of this, Bootstrap tackles the problem of inconsistent form input control appearance across different browsers, by offering custom check boxes and radio inputs.

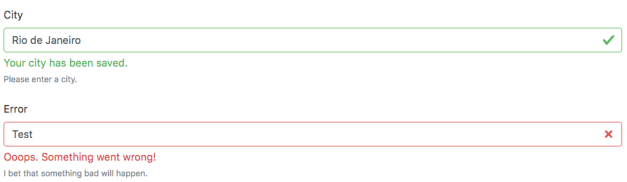


Figure 2: An example of Bootstrap's input validation styles: has-success and has-error will cause corresponding feedback icons to be displayed, along with applying the correct context colours

By applying the `custom-control` class along with either the `custom-checkbox` or `custom-radio` class to the parent element of any input element that has been assigned the `custom-control-input` class and which contains a sibling with the `custom-control-indicator` class, Bootstrap will ensure consistent render across different browsers:

```
<label class="custom-control custom-
checkbox">
  <input class="custom-control-input"
  name="checkbox" type="checkbox" >

  <span class="custom-control-
  indicator"></span>
  <span class="custom-control-
  description">My checkbox</span>
</label>
```

Figure 3: An example of custom radio and checkbox controls using Bootstrap 4.

Navigation

Another nice change that ships with Bootstrap 4 is the simplification of the navigation component. In order to get navigation working in Bootstrap 3, one would need to create a list of elements, using the `nav` base class. One would then need to apply the `nav-tabs` class, and necessary styling to the tab headings. This was somewhat unintuitive, and produced slightly messy markup:

```
<ul class="nav nav-tabs nav-justified"
role="tablist">
  <li role="presentation"
  class="active">
  <a href="#my-id" role="tab" data-
  toggle="tab">Foobar</a>
  </li>
</ul>
```

In contrast, with Bootstrap 4's new navigation classes, this markup becomes clearer and easier to read:

```
<ul class="nav nav-tabs nav-justified">
  <li class="nav-item">
    <a href="#my-id" data-toggle="tab"
    class="nav-link active">Foobar</a>
  </li>
</ul>
```

That is, Bootstrap 4 introduces: i) the `nav-item` class for denoting navigation elements, ii) the `nav-link` class for denoting the actual anchor elements. Furthermore, it requires the anchor element, not the nav item element, to be used when denoting whether a link is `active` or not.

Another notable addition when it comes to navigation, are the navigation bar styles: `navbar-dark` and `navbar-light`. These two classes simply set the color of the navbar to white or black, in order to support a dark navbar background or a light navbar background. This allows one to easily change the appearance of the navbar by applying context classes. For example:

```
<nav class="navbar navbar-dark bg-
danger">
</nav>
<nav class="navbar navbar-light bg-
success">
</nav>
```

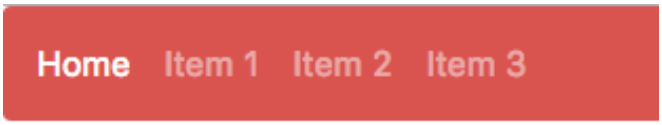


Figure 4: A sample navbar with navbar-dark and bg-danger applied.

Last but not least, Bootstrap 4 ships with a new button style: `btn-outline-*` (where * denotes the context style) for creating button outlines. For example, to create a `primary` button outline, one would apply the `ban-outline-primary` class.

Figure 5: A normal button using btn-primary (left) and a button outline using the btn-outline-primary class.

Dropdowns

Similar to navigation items, every dropdown item now requires that the `dropdown-item` class be applied to it. Furthermore, the caret symbol is now

automatically applied to dropdown toggles that have been denoted as such using the `dropdown-toggle` class. The markup for a typical dropdown navigation menu item in Bootstrap 4 would look as follows:

```
<div class="dropdown">
  <button class="btn btn-primary
  dropdown-toggle" type="button" data-
  toggle="dropdown" aria-haspopup="true"
  aria-expanded="false">

  My Dropdown
  </button>
  <div class="dropdown-menu">
    <a class="dropdown-item" href="#">My
      Menu Item</a>
  </div>
</div>
```

Pagination

In line with the changes introduced for the navigation bar and dropdowns, the markup required to produce a pagination component has also been clarified: unlike Bootstrap 3, Bootstrap 4 now requires one to explicitly denote the pagination and link items items using the `page-item` and `page-link` classes respectively. That is, in Bootstrap 3, we could simply create a pagination component by applying the `pagination` class to any `ul` element:

```
<ul class="pagination">
    <li><a href="#">1</a></li>
    <li><a href="#">2</a></li>
</ul>
```

In Bootstrap 4, we must instead denote the individual pagination item and links:

```
<ul class="pagination">
  <li class="page-item"><a class="page-
  link" href="#">1</a></li>
  <li class="page-item"><a class="page-
  link" href="#">2</a></li>
</ul>
```

Browser support

Bootstrap 4 supports all major browsers, including Chrome, Safari, Opera and any version of Internet Explorer greater than 8.

Notable classname changes

In addition to the previously discussed feature additions, removals and/or changes, Bootstrap 4 Alpha introduces a bunch of notable class name changes to watch out for when migrating:

- The `btn-default` class name has changed to `btn-secondary`.
- Extra small buttons no longer exist, as the `btn-xs` class has been removed. Likewise, extra small button groups no longer exist through the removal of `btn-group-xs`.
- The `divider` class used as part of dropdowns has been renamed to dropdown-divider.
- The `navbar-form` class has been removed.
- The `label` has been replaced with the tag class.
- The `Carousel` item class has been renamed to `carousel-item`.
- The offsetting of columns has changed, from `.col-*-offset-** to .offset-*-**.` So, for example, `.col-md-offset-2` becomes `.offset-md-2`. Similarly, the pushing of columns changed from `.col-push-*-*` to `.push-*-*`.

Plugins

There are no nasty surprises or drastic changes in how Bootstrap plugins are composed: As with Bootstrap 3, plugins are split across two files - a JavaScript file, and a file containing style rules. In the case of Bootstrap 4, these style rules are of course written using Sass. For example, the Bootstrap button plugin consists of: bootstrap/js/src/button.js and bootstrap/scss/mixins/_buttons.

scss. As Bootstrap is compiled, they are integrated as part of the Bootstrap distributable, and as such, the end-user needs to spend little to no time worrying about them.

The JavaScript file typically contains the class definition that defines the actual plugin, and a data API implementation which hooks on the DOM, and listens for actions. For example, the data API implementation of Bootstrap's button plugin looks as follows:

```
$(document)
  .on(Event.CLICK_DATA_API, Selector.
DATA_TOGGLE_CARROT, (event) => {
  event.preventDefault()

  let button = event.target

  if (!$(button).hasClass(ClassName.
  BUTTON)) {
  button = $(button).closest(Selector.
  BUTTON)
}

  Button._jQueryInterface.
  call($(button), 'toggle')
})
  .on(Event.FOCUS_BLUR_DATA_API,
  Selector.DATA_TOGGLE_CARROT, (event)
  => {
    let button = $(event.target).
    closest(Selector.BUTTON)[0]
    $(button).toggleClass(ClassName.
    FOCUS, /^focus(in)?$/.test(event.
     type))
  })
```

Last but not least, the plugin's JavaScript file contains a jQuery section, which makes the plugin available to the global jQuery object:

```
$.fn[NAME]= Button._jQueryInterface
  $.fn[NAME].Constructor = Button
  $.fn[NAME].noConflict  = function () {
    $.fn[NAME] = JQUERY_NO_CONFLICT
    return Button._jQueryInterface
}
```

Whilst adding the plugin to the global jQuery object, one can use any non-conflicting name. In the case of the button plugin, this name is simple the string button.

Utility classes

With Bootstrap 4 comes a nice addition of various new utility classes, without breaking any of the existing major functionality. For one, the text alignment classes remain the same, with a the nice addition of responsive text alignment classes: `text-*-left, text-*-center` and `text-*-right`. For example, to center text on medium (MD) viewports, one would use the `text-md-center` class.

The new utility classes also provide various shortcuts that allow you to avoid having to set inline styles or write your own classes: the `d-*` classes allow you to set the display property to either `block, inline` or `inline-block`. For example, `d-block` will set the display property of an element to `block`, whilst `d-inline-block` will set it to `inline-block`. Similarly, the `w-*` classes, allow you to quickly set the width of an element to the desired %. As such, `w-100` would set the width property of an element to 100%, whilst `w-50` would set it to 50%. Similarly, the Bootstrap 4 comes with short-hand rules for setting the margin or padding of an element. The classes take the form of `<property abbreviation>-<side abbreviation>-<integer size>`. Within this context, the allowed property abbreviations are `m` (margin) and `p` (padding), whilst the permissible side abbreviations are `t` (top), `b` (bottom), `l` (left), `r` (right), `a` (for setting both top, bottom, left and right), `x` (for setting both left and right) and `y` (for setting both top and bottom). The size is an integer in the range of 1 - 3, inclusive. So, for example, to set the left margin of an element to 0, one would use the class `m-l-0`.

Another nice addition are responsive floats: Instead of just being able to use `pull-left` and `pull-right`, Bootstrap now allows for responsive floats, depending on the viewport size by providing the classes `pull-*-right` and `pull-*-left`.

Last but not least: the new utility classes also include support for responsive embedding. Using responsive embedding, the aspect ratio of embedded media will scale with the viewport size. Simply apply the `embed-responsive-item` class to any `iframe`, `object`, `video` or `embed object`.

Ready to try Bootstrap?

If you are now ready to explore Bootstrap 4 Alpha on your own, then head over to http://v4-alpha.getbootstrap.com.

Alternatively, you can install Bootstrap 4 Alpha using nam or bower:

```
npm install bootstrap@4.0.0-alpha.4
```

or

```
bower install bootstrap#v4.0.0-alpha.4
```

Conclusion

Being a complete re-write, Bootstrap 4 differs to its predecessor in many ways. Luckily, the vast majority of differences are subtle and none-breaking. This article covered some of the most obvious differences, highlighting the most anticipated new features ■

About the Author

benjamin Jakobus

Benjamin Jakobus graduated with a BSc in Computer Science from University College Cork and obtained an MSc in Advanced Computing from Imperial College London. As a software engineer, he has worked on various web-development projects across Europe and Brazil.

A MAGAZINE FOR .NET AND JAVASCRIPT DEVS

- ☑ AGILE
- ☑ ASP.NET
- ☑ MVC, WEB API
- ☑ ANGULAR.JS
- ☑ NODE.JS
- ☑ AZURE
- ☑ VISUAL STUDIO
- ☑ .NET
- ☑ C#, WPF

We've got it all!

100K PLUS READERS

230 PLUS AWESOME ARTICLES

25 EDITIONS

FREE SUBSCRIPTION USING YOUR EMAIL

EVERY ISSUE DELIVERED

RIGHT TO YOUR INBOX

NO SPAM POLICY

SUBSCRIBE TODAY!

Gouri Sohoni

Code Quality Tools in Visual Studio 2015

Code Quality is a loose approximation of writing useful, correct and maintainable code. Writing good quality code helps in reducing bugs at a later stage in product development. Visual Studio provides many built-in tools for writing quality code.

This article will discuss various tools available for developers for testing using Visual Studio. There are tools like Code Analysis, Code Metrics, IntelliTrace, and Code Profiling which can be used to deliver high quality code. Writing unit tests, using IntelliTest, finding Code Coverage will also result in writing qualitative code. We will also see how to write Unit Test with MS Framework, third party testing, as well as writing IntelliTest. Later we will discuss how Code Coverage helps in unit testing and take a brief overview of Code Analysis, Maintainability Index, IntelliTrace and Code Profiling

Once the code is written, unit testing helps developers find logical errors in the code. In some cases, Test Driven Development (TDD) approach is used. In TDD, the test is written first and then the code is written for the test method.

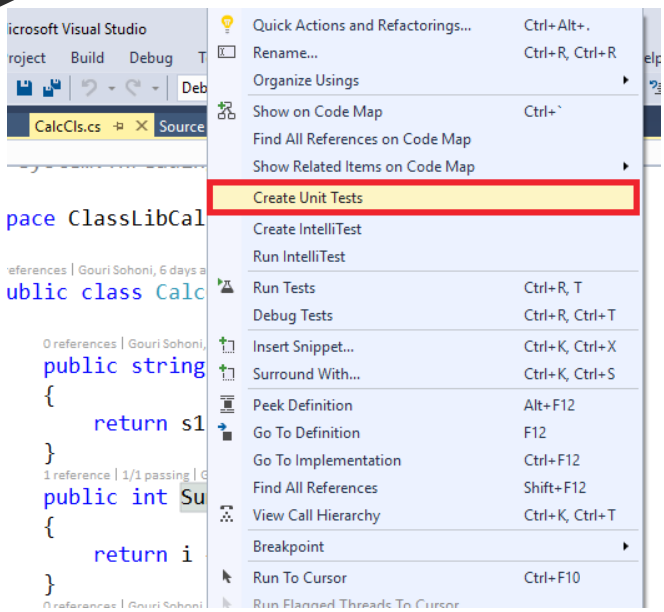
We will discuss how to create unit test for code which is already available. Download and install Visual Studio 2015 if you haven't already done so.

Testing Related Tools for Developers

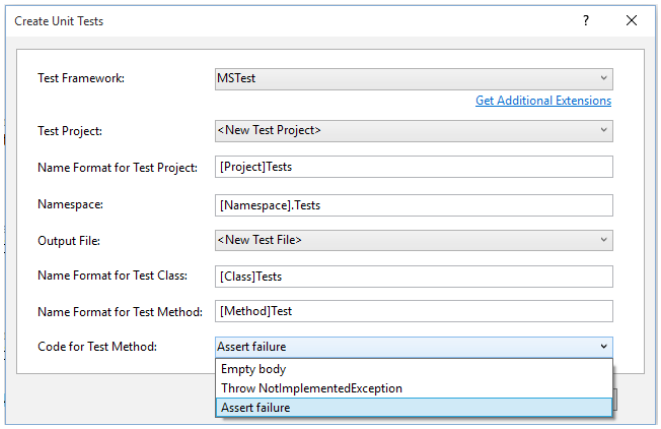
Creating Unit Tests

1. Create a solution and add a class library with some functionality to it. We will write unit tests for the methods in the class library.

2. Add Unit Test Project to the current solution. Select a method for which we need to create a unit test, right click and select Create Unit Tests



3. Select the option of new Test Project. The Unit Test Project is added, reference set to the class library and you can select Empty body, throw exception or add the statement for Assert.Failure as shown in this figure.



4. For this example, select the default Assert failure statement and the stub for Test Method looks as follows

```
[TestClass()]
0 references
public class CalcClsTests
{
    [TestMethod()]
    0 references
    public void SumTest()
    {
        Assert.Fail();
    }
}
```

5. The class gets an attribute as TestClass, and the method gets attribute as TestMethod. Without any of these attributes, the test method will be ignored.

6. Let us add code to test the method. You can run the test using Test Explorer. If it is not visible, go to Test – Windows – Test Explorer to view it. Select the test method, right click and choose run.

```
[TestMethod]
[PexGeneratedBy(typeof(CalcClsTest))]
0 references
public void Div248()
{
    int i;
    CalcCls s0 = new CalcCls();
    i = this.Div(s0, 0, 1);
    Assert.AreEqual<int>(0, i);
    Assert.IsNotNull((object)s0);
}
```

The test passes.

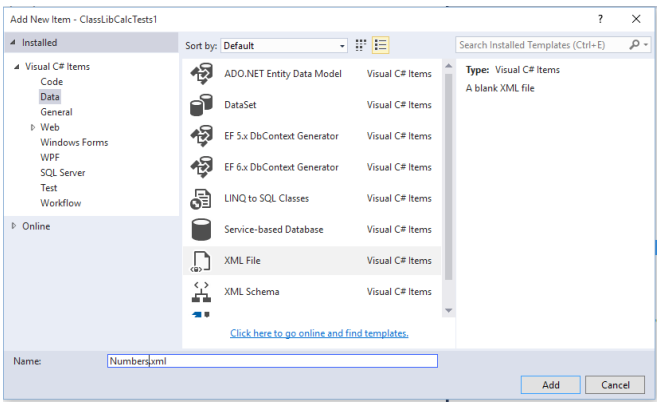
7. There are four additional attributes for initializing and cleaning up for class or test. *ClassInitialize* and *ClassCleanup* methods will be executed when you run the first test in the class, and when you finish running the last test in the class. Similarly *TestInitialize* and *TestCleanup* methods are called before test execution and after test running.

```
[ClassInitialize()]
0 references
public static void MyClassInitialize(TestContext testContext) { }

[ClassCleanup()]
0 references
public static void MyClassCleanup() { }

[TestInitialize()]
0 references
public void MyTestInitialize() { }

[TestCleanup()]
0 references
public void MyTestCleanup() { }
```

Convert Unit Test to Data Driven Unit Test

In the previous example, the test method took the same set of parameters and gave the same result. In a real life scenario, it's better to change parameters on the fly. In order to achieve this, we need to convert this test method to data driven test. The data can be provided via xml file, csv file or even via database.

1. Add a New Item, select Data tab and select XML file. Change the name of the XML file to anything you wish. Go to properties of the file, and change the Copy to Output Directory to Copy if newer



2. Provide data for parameters along with the expected result.

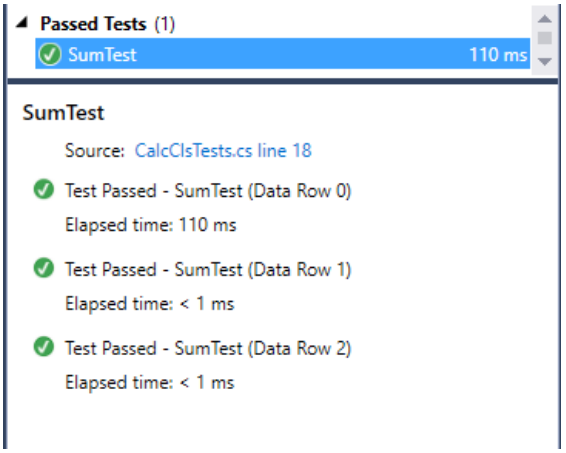
```
Numbers.xml | CalcClsTests.cs | CalcClsTests.cs | CalcCls.cs
<?xml version="1.0" encoding="utf-8" ?>
<data>
  <numbers>
    <num1>10</num1>
    <num2>20</num2>
    <result>30</result>
  </numbers>
  <numbers>
    <num1>20</num1>
    <num2>40</num2>
    <result>60</result>
  </numbers>
  <numbers>
    <num1>1</num1>
    <num2>2</num2>
    <result>3</result>
  </numbers>
</data>
```

3. Now we need to add a DataSource parameter to the method, the code looks as follows. Add a reference to System.Data assembly, and add the code for TestContext.

```
[TestClass()]
public class CalcClsTests
{
   [DataSource("Microsoft.VisualStudio.TestTools.DataSource.XML",
      "|DataDirectory|\\Numbers.xml","numbers",DataAccessMethod.Sequential),
       TestMethod()]
    public void SumTest()
    {
        CalcCls target = new CalcCls();
        int actual, expected = Int32.Parse(TestContext.DataRow["result"].ToString());
        actual = target.Sum(Int32.Parse(TestContext.DataRow["num1"].ToString()),
            Int32.Parse(TestContext.DataRow["num2"].ToString()));
        Assert.AreEqual(expected, actual);

    }
```

```
private TestContext testContextInstance;

3 references | 1/1 passing | 0 changes | 0 authors, 0 changes
public TestContext TestContext
{
    get
    {
        return testContextInstance;
    }
    set
    {
        testContextInstance = value;
    }
}
```

4. Run the test and observe that it gets executed three times, same as the number of records in the data file.

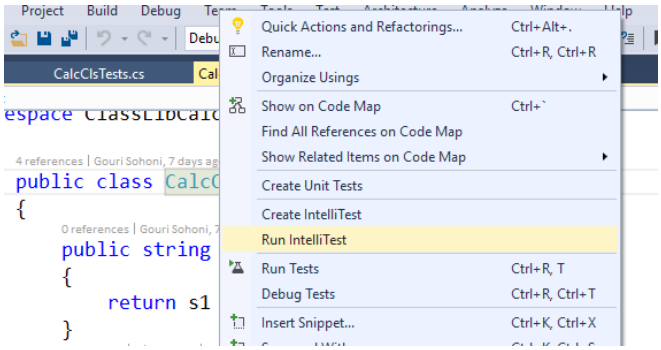


A Data Driven Unit test can pass different set of parameters every time it executes.

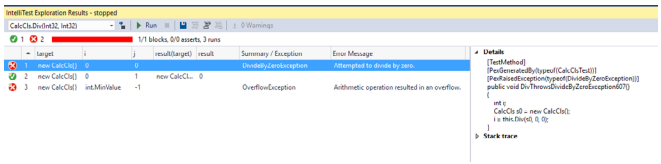
IntelliTest

Visual Studio offers a useful utility in the name of IntelliTest (formerly called as SmartTest). With this tool, you can find out how many tests are passing or failing. You can also provide the code to fix issues. Writing an exhaustive test suite for a very complex piece of code, requires a lot of efforts. There is a tendency to omit some test data which may lead to bugs getting captured at a far later stage. IntelliTest takes care of this problem. It will help in early detection of bugs, and lead to better qualitative code.

1. Right click on the class and select Run *IntelliTests* option



2. It will create a Test Suite for all the methods in the class, and generate data. Every code is analysed depending upon any *if* statements, loops. It shows what kind of exceptions will be thrown.



If we select the test case, it shows us the details and how code can be added. In this example, Divide by zero exception can be handled by adding an attribute. We can add the necessary code.

3. We have the option of actually creating a Test Project and adding all the test methods to it. IntelliTest adds Pex attributes to the code, as can be seen from following image

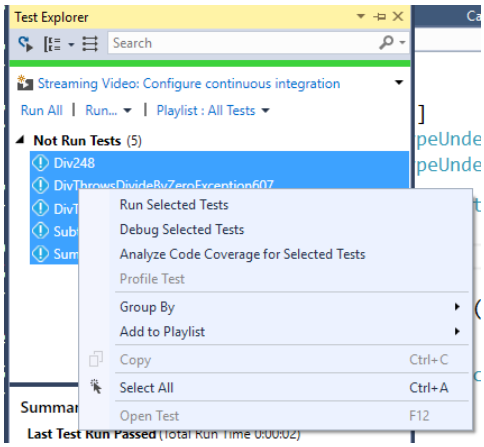
```
[TestMethod]
[PexGeneratedBy(typeof(CalcClsTest))]
public void Div248()
{
    int i;
    CalcCls s0 = new CalcCls();
    i = this.Div(s0, 0, 1);
    Assert.AreEqual<int>(0, i);
    Assert.IsNotNull((object)s0);
}
```

IntelliTest when created, tries to find the path for high code coverage. Let us find out how Code Coverage will help in increasing Code Quality.

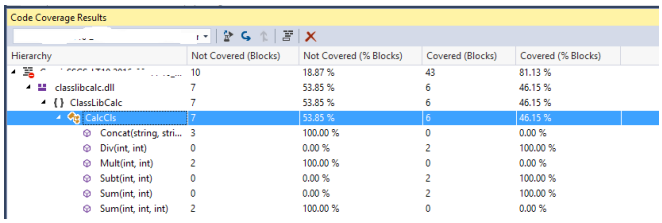
Analyzing Code Coverage

Code Coverage determines which code is getting executed when the unit test is run. If there are complex and multiple lines of code, Code Coverage will help in finding out if a portion of the code is not being tested at all. We can take necessary action based on these findings. Code coverage can be computed when you are executing tests using Test Explorer.

1. From Test Explorer, select the unit tests to be executed, right click and select "Analyse Code Coverage for Selected Tests"

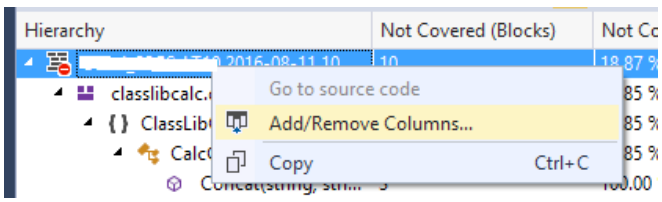


2. The Code Coverage Results can be seen as follows:

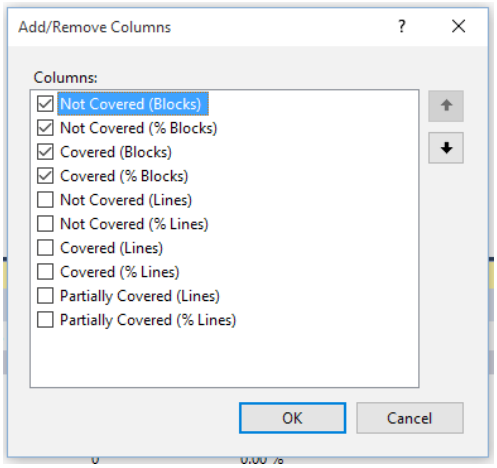


3. If we navigate to the actual assembly, we can see the blue and red coloured code which indicates if the code was executed or not.

4. Code Coverage is measured in blocks. A block is a code which has only one entry and exit point. If the control passes through a block, it is considered as covered. You can also add columns by right clicking the assembly, and selecting Add/Remove Columns



5. Select lines covered with their percentage.



6. The complete data will be shown with Code Coverage Results.

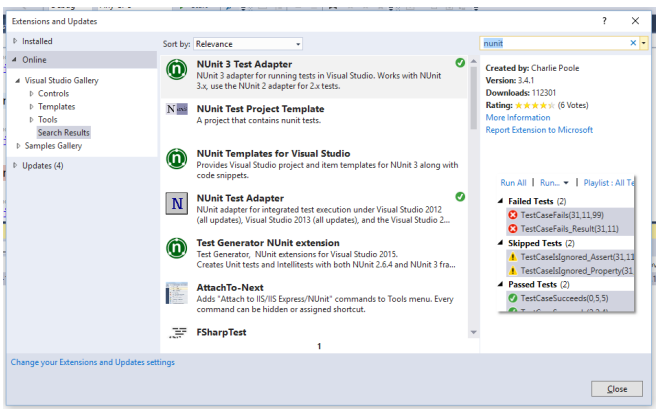
7. In order to view the previous results, Import the results. To send the results to someone, export it.

Third Party Testing Framework

At times, we can use a third party testing framework for unit testing. The default testing framework available with Visual Studio is MSTest. You can download and install various third party testing framework with Visual Studio.

1. Select Tools > Extensions and Updates and Online > Visual Studio Gallery tab.

2. Type the name of the framework in the search box.



3. Download and install the NUnit framework.

4. Select the Test Project, right click and select Manage NuGet Packages. Find NUnit and Install. A reference to nunit.framework.dll gets automatically added.

5. Add a new Item of type unit test, and change the code to the following:

```
using NUnit.Framework;

namespace ClassLibCalc.Tests
{
    [TestFixture]
    1 reference
    public class nUnit
    {
        0 references
        public nUnit()...

        Additional test attributes

        [Test]
        0 references
        public void MultTest()
        {
            CalcCls target = new CalcCls();
            int actual, expected = 40;
            actual = target.Mult(20, 2);
            NUnit.Framework.Assert.AreEqual(actual, expected);
        }
    }
}
```

6. Execute test from the Test Explorer.

Until now, we have seen all the testing related tools for developers. These tools help in identifying bugs at an early stage thereby improving overall code quality.

Now let us delve into tools which will help developers write better quality code.'

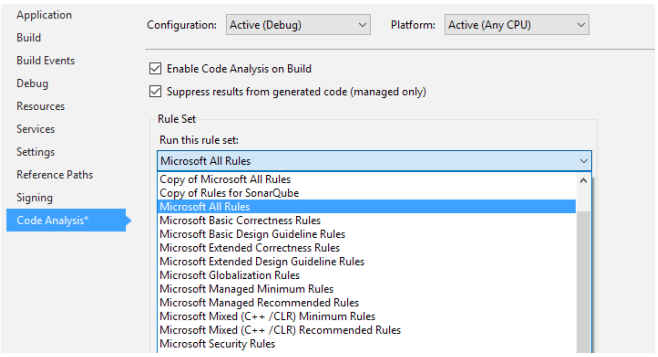
Tools to write Quality Code

Code Analysis

The code being written needs to follow certain rules. The rules can be according to the organization's standards, or certain standards enforced by the customer for whom the application is being developed. Code Analysis (also called as Static Code Analysis) helps in finding out if there are areas in the code not following a set of

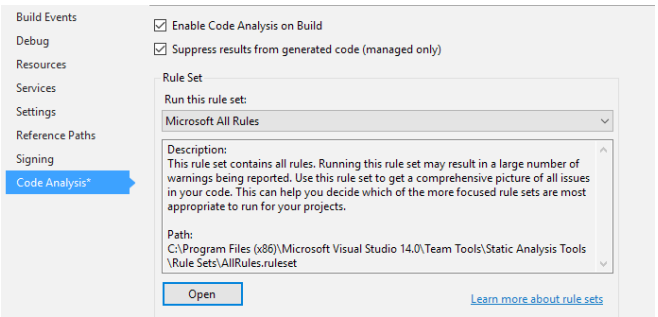
accepted rules. This tool is a part of Visual Studio, and can be applied to any project. We can specify that code analysis is to be enabled at the time of build, or it can be applied as and when required. Visual Studio provides a lot of rule sets like Microsoft All Rules, Microsoft Basic Correctness Rules, and Microsoft Basic Design Guideline Rules etc. Using these rulesets, either one or multiple rulesets can be applied to a project. The default behaviour gives warnings if the rules are not adhered to.

1. In order to see the rulesets, right click on the project to apply code analysis, and select properties. Select the tab for Code Analysis and click on the drop down to get a list of rulesets



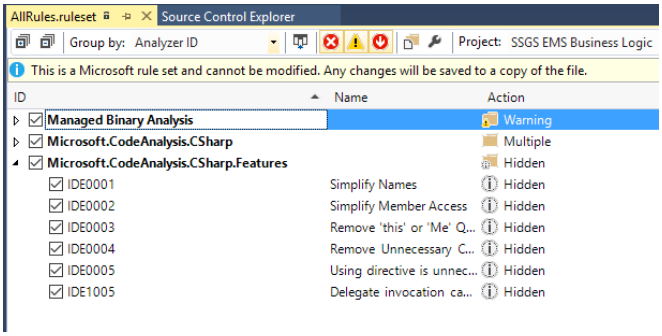
2. You can also right click on project and select Analyze > Run Code Analysis.

Click on Open to open the selected ruleset.

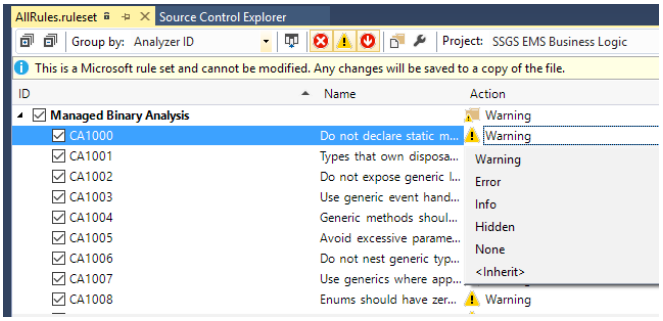


Observe the check box to *Enable Code Analysis on Build*.

3. Various rules are displayed for each category. We can change any of these rules and create a copy of the ruleset. We cannot modify the existing ruleset, but the copied one can be edited as required. Open the ruleset and expand one of the categories.



4. Actions can be applied to any rule.



For most of the rules, we get a warning if the rule is not followed. By changing it to error, we can ensure that the developer cannot ignore the rule, as its a human tendency to ignore any warnings.

5. After changing one or multiple rules in a ruleset, we can provide a name for it, and save this copy of ruleset for future reference. Later this set can be applied to a project, to get the errors and or warnings as set earlier.

Note: Code Analysis was previously known as FxCop. This tool helps in finding out if there are any programming related issues in the code.

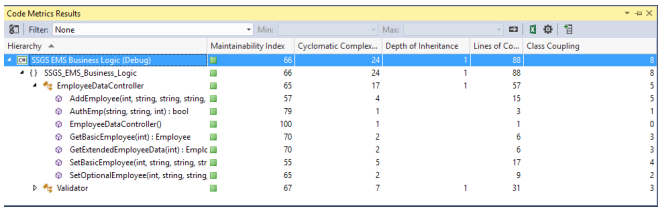
Code Metrics

Visual Studio can measure maintainability index for the code. It also finds the complexity of the code. As the name suggests, Code Metrics provides information if the code is maintainable or not. Originally this algorithm was developed by Carnegie-Mellon University. Microsoft adopted this and incorporated it as a part of Visual Studio from Visual Studio version 2008 onwards. It evaluates

code based upon four criterias - Cyclomatic Complexity, Depth of Inheritance, Class Coupling and Lines of Code.

Cyclomatic Complexity is based on loops and various decisions in code. **Class Coupling** finds the number of dependencies on other classes. More the class coupling, lower is the index. **Depth of Inheritance** is for inheritance of classes from the Object class. **Lines of code** is the actual number of executable lines. The index is between the range of 0 to 100. 0 to 9 is low, 10 to 19 is moderate and 20 onwards is high. The higher the maintainability index, the better are chances of maintaining it.

For using this tool, you only need to select the project or complete solution, right click and select Calculate Code Metrics. The results will be shown with the classification of parameters mentioned.

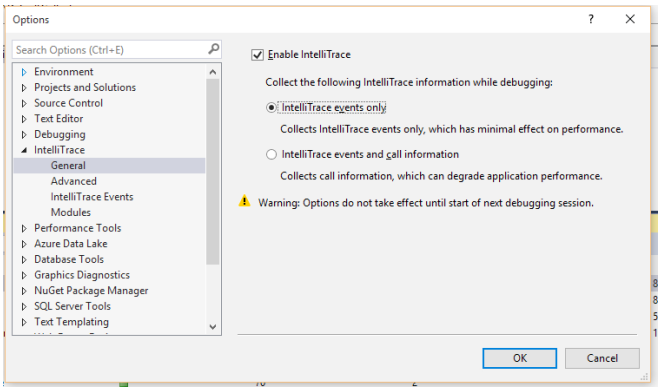


IntelliTrace

This tool was introduced by Microsoft in Visual Studio 2010. It was originally called as *historical debugger* as it gives history about the debugging parameters and variables. Every developer needs to debug as a part of the routine. Sometimes it takes a lot of time to go through the steps till the breakpoint is hit. If a developer doesn't know the code well or doesn't remember the breakpoints, then debugging can prove to be a very time consuming process.

IntelliTrace helps in minimizing the time for debugging.

By default, IntelliTrace is enabled, we can select it from Tools > Options and selecting IntelliTrace tab



There are 2 ways with which IntelliTrace information can be gathered - by collecting only events information, or collecting events and calls information. We can also select the event information that needs to be captured. The more the number of events captured, the more information we get, but the overall performance of the application will be hampered.

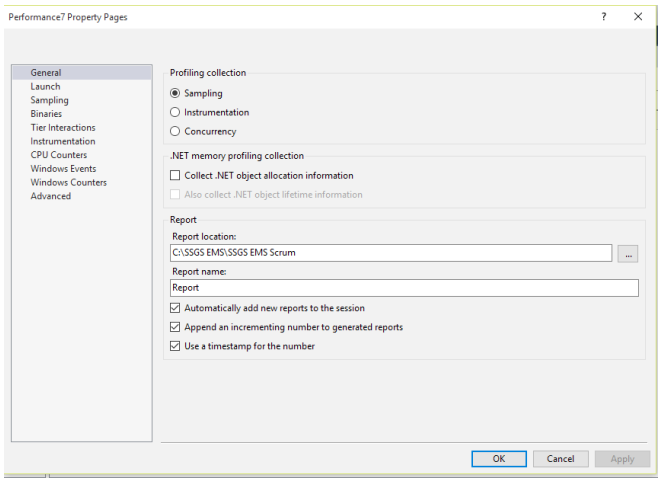
Intellitrace creates .ITrace file which will store complete information about events and calls. This file gets automatically deleted once the instance of Visual Studio is closed. We can also change the location of .ITrace file and store it for further usage. This file can be opened by Visual Studio, and will give a complete history of debugging to your colleague in case you need any help.

IntelliTrace file can be created by using Microsoft Test Manager, and also at production level.

Code Profiling

We use Code Profiling for finding out performance related issues in an application. It is mandatory to have .pdb files while profiling code. There are three ways with which performance data can be collected. These are Sampling, Instrumentation and Concurrency.

You can also use Sampling or Instrumentation method to collect data about memory allocation. To start profiling, you can use the performance wizard which can be started from Debug > Performance > Performance Explorer > New Performance Session. The properties of the performance session will provide collecting parameters.



Conclusion

This article discussed various Testing Tools from a Developer's perspective. We discussed how to create unit tests and convert it to data driven test. Data Driven Tests are useful for executing multiple sets of parameters, and find the result. We discussed the importance of IntelliTest, how to include and write unit tests for third party framework. We also discussed how to find out the code covered during execution of tests. We have seen how tools like Code Analysis, Code Metrics help in writing better quality code. IntelliTrace helps in debugging and gives us historical data for debugging. The Code Profiling tools help in analysing performance issues for an application.

These set of tools in Visual Studio 2015 can provide developers better insights into the code they are developing, and help them identify potential risks and fix them, in order to create quality, maintainable code

About the Author

Gouri Sohoni

Gouri Sohoni is a Microsoft MVP, Trainer and consultant for over two decades. She is a Microsoft Certified Trainer (MCT), MCITP and has conducted several corporate trainings on Microsoft technologies that include Visual Studio 2010 (ALM), SQL Server 2008 BI, SQL Server 2008 developer track, SQL Server 2005 BI, SQL Server 2005 developer track etc

A MAGAZINE FOR .NET AND JAVASCRIPT DEVS

- AGILE
- ASP.NET
- MVC, WEB API
- ANGULAR.JS
- NODE.JS
- AZURE
- VISUAL STUDIO
- .NET
- C#, WPF

We've got it all!

100K PLUS READERS

230 PLUS AWESOME ARTICLES

25 EDITIONS

FREE SUBSCRIPTION USING YOUR EMAIL

EVERY ISSUE DELIVERED

RIGHT TO YOUR INBOX

NO SPAM POLICY

SUBSCRIBE TODAY!

THANK YOU

FOR THE 26th EDITION

@damirarh

@vikrampendse

@gouri_sohoni

@dani_djg

bit.ly/dnc-rahul

@yacoubmassad

@benjaminjakobus

@suprotimagarwal

@saffronstroke

WRITE FOR US