ISSUE 33 NOV-DEC 2017

DNCMAGAZINE

www.dotnetcurry.com

C# 7.x AND BEYOND

Angular Application Architecture

ASP.NET Core 2.0 - What's New

NEW FEATURES IN

Visual Studio 2017

The Producer-Consumer pattern in .NET

DynamicQueue
Seamless Communication in C#

ECMASCRIPT 2017 (ES8) - WHAT'S NEW

FLUX VS MVC

FROM THE EDITOR

At the recently concluded .NET Conf 2017 in mid-September, Microsoft showcased some cool cross-platform apps that can be build with .NET. From platforms and tools like .NET Core, ASP.NET Core, Visual Studio 2017 and Azure to programming languages such as C# and F#, this virtual conference offered a glimpse at the direction these technologies are moving towards.

This month's issue of the DNC Magazine focuses on some of the platforms and technologies that was covered in .NET Conf.

In this edition, Damir covers some upcoming features in the next version of C# 7.X and 8.0, while Mahesh covers some cool productivity features in VS 2017. Daniel explores some big and small improvements in ASP.NET Core 2.0. For our JavaScript fans, Keerti does an overview of what's new in the latest version of JavaScript - ES8, while Ravi explains a high level architecture of an Angular application. First time DNC author Muhammad shows off his DynamicQueues framework and Rahul compares two popular design patterns - Flux and MVC. Last but not the least, Yacoub talks about the Producer-Consumer Pattern and how to correctly use it in .NET.

Editor in Chief

Suprotim Agarwal

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

THE TEAM

Editor In Chief
Suprotim Agarwal

Art Director
Minal Agarwal

Contributing Authors
Damir Arh
Daniel Jimenez Garcia
Keerti Kotaru
Mahesh Sabnis
Muhammad Ahsan
Rahul Sahasrabuddhe
Ravi Kiran
Yacoub Massad

Technical Reviewers
Damir Arh
David Pine
Keerti Kotaru
Ravi Kiran
Suprotim Agarwal
Yacoub Massad

Next Edition
January 2018

THANK YOU

FOR THE 33rd EDITION

@damirarh

@yacoubmassad

@sravi_kiran

@dani_djg

@maheshdotnet

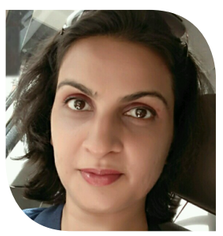
@rahul1000buddhe

@keertikotaru

@davidpine7

muhammad ahsan

@suprotimagarwal



@ saffronstroke

WRITE FOR US

mailto: suprotimagarwal@dotnetcurry.com

Introducing **RavenDB 4.0**
Your Fully Transactional NoSQL Database

The amount of data your organization needs to handle is rising at an ever-increasing rate. We developed RavenDB 4.0 so you can handle this tougher challenge, and do it while improving the performance of your application at the same time.

RavenDB 4.0 is the premiere choice of Fortune 500 companies because it offers you the best of both worlds. It is a NoSQL database that is fully transactional. You can get the benefits of using next generation NoSQL while keeping the best value that relational databases offer. Our open source document database has reached over 100,000 writes and half a million reads per second using low cost commodity hardware. We ramp up your performance right up until you hit the limits of your hardware.

It's easy to set up a RavenDB 4.0 database cluster and even easier to use. The ramp up time to become an expert is quick. Our RQL query language is similar to SQL, and very familiar with what you are used to. The management studio GUI makes using RavenDB 4.0 convenient for both developers and non-developers. We've automated many of the functions normally assigned to DBAs, freeing up time and resources for other priorities.

Your data never sleeps, so why should your database? Our multi-node data cluster eliminates the single point of failure vulnerability that comes with the relational SQL approach. You can create a distributed data cluster with multiple nodes, all holding a copy of your database and replicating to each other in real time. This gives you high-availability while effectively balancing load, reducing latency, and maximizing your performance. Your customers have the technology to seek information about your business 24 hours a day. With a RavenDB 4.0 data cluster you have the ability to give it to them anytime they want.

Our in-house storage engine pushes performance to the max and minimizes the need for third party plugins. With everything at your fingertips, RavenDB 4.0 gives you all the tools to take your database technology to the next level.

Enjoy Top Performance

Fully Transactional
like a relational database

Easy to Install
Easy to Use

Works Well Alongside SQL Solutions

Scale Up Fast with a High Availability Data Cluster

All-in-One Database
Fewer third party plugins
puts everything at your fingertips

Highly Automated Features
to reduce overhead

Hibernating Rhinos

RAVENDB 4.0
DATA MADE FASTER

Grab a FREE License
3-node database cluster with GUI interface,
3 cores and 6 GB RAM

www.ravendb.net/free

GdPicture.NET 14

100% ROYALTY FREE
Imaging SDK For **WinForms**, **WPF** And **Web** Development

TWAIN
OCR
100+ Supported Formats
JBIG2
ANNOTATIONS
MICR
OMR
Thumbnail Control
Image Viewer
PDF Toolkit
Color Detection
Printing
DICOM
TIFF
DOCX
NEW RELEASE

Leverage your apps. with **GdPicture.NET** Imaging Toolkit

DOWNLOAD YOUR FREE TRIAL

www.gdpicture.com

GdPicture.NET is an ORPALIS imaging technologies product

TABLE OF
CONTENTS

Yacoub Massad

The Producer-Consumer Pattern in .NET

This article discusses the producer-consumer pattern in .NET. I will discuss the reasons why we would use it and show some examples of how to implement it in .NET.

INTRODUCTION

As a result of machines having multiple processing cores, parallel programming is becoming more important these days. A typical PC these days has four to eight cores, and servers have a lot more than that.

It makes sense to write software that uses multiple cores to enhance performance. Consider the following code from an application that processes documents:

As a result of machines having multiple processing cores, parallel programming is becoming more important these days. A typical PC these days has four to eight cores, and servers have a lot more than that.

It makes sense to write software that uses multiple cores to enhance performance.

Consider the following code from an application that processes documents:

```
void ProcessDocuments()
{
  string[] documentIds = GetDocumentIdsToProcess();

  foreach(var id in documentIds)
  {
    Process(id);
  }
}

string[] GetDocumentIdsToProcess() => ...

void Process(string documentId)
{
  var document = ReadDocumentFromSourceStore(documentId);

  var translatedDocument = TranslateDocument(document, Language.English);

  SaveDocumentToDestinationStore(translatedDocument);
}

Document ReadDocumentFromSourceStore(string identifier) => ...

Document TranslateDocument(Document document, Language language) => ...

void SaveDocumentToDestinationStore(Document document) => ...
```

This code gets the ids of documents that need to be processed, obtains each document from the source store (e.g. database), translates it, and then saves it to the destination store.

This code right now will run under a single thread.

In C#, we can easily modify this code to use multiple cores using the `Parallel` class like this:

```
void ProcessDocumentsInParallel()
{
  string[] documentIds = GetDocumentIdsToProcess();
  Parallel.ForEach(
    documentIds,
    id => Process(id));
}
```

This is called Data Parallelism because we apply the same operations on each data item; and in this particular case, on each document id.

`Parallel.ForEach` will parallelize the invocation of the `Process` method over the items in the `documentIds` array. Of course, all of this depends on the environment. For example, the higher the number of available cores, the more the number of documents that can be processed in parallel.

Consider the following figure:

Assuming that currently eight threads are processing, we can imagine each of these threads taking one document id, calling `ReadDocumentFromSourceStore`, `TranslateDocument`, and then `SaveDocumentToDestinationStore`.

Once each thread is done with its document, it will grab another document id and process it. `Parallel.ForEach` will manage all of this.

Although this might seem a good idea, there might be some issues with this approach.
I will go through these issues and finally introduce the producer consumer pattern and show how it solves some of these issues.

Note: *The behavior of Parallel.ForEach (and the .NET components used by it) is sophisticated. It automatically manages the degree of parallelism. For example, even if a machine has eight cores, the system might decide to run more than eight tasks in parallel if it sees that the currently executing tasks, are blocked by I/O. To keep the discussion simple, I am going to assume that in a machine with eight cores, Parallel.ForEach will have exactly eight tasks running in parallel.*

Simple parallelism – Some Issues

Too much parallelism for some operations

In the previous example, we assumed that the system will use eight thread-pool threads to process the documents. This means that it is possible that at a given instance, there would be eight threads invoking `ReadDocumentFromSourceStore`.

This might be problematic if `ReadDocumentFromSourceStore` is an I/O operation on a device that performs badly on parallel access.

For example, `ReadDocumentFromSourceStore` might be reading the document from a simple Hard Disk Drive (HDD). When reading from the HDD in parallel, the disk might to need to read data from different locations. Moving from one location to another in HDDs is slow because the physical head in the disk needs to move.

On the other hand, if we are reading a single file at a time, it is more likely that the contents of the file (or large segments of the file) are in the same physical location on the disk and therefore the disk takes less time moving the head. You can read more about hard disk performance here: https://en.wikipedia.org/wiki/Hard_disk_drive_performance_characteristics

To fix this issue, we can put a lock or a semaphore around the invocation of the `ReadDocumentFromSourceStore` method.

A lock would allow only one thread to invoke this operation at a time, and a semaphore will allow a predefined number of threads (specified in code) to invoke the method at a time. The following example uses a semaphore to make sure that at any point of time, at most two threads will be invoking `ReadDocumentFromSourceStore`.

```
Semaphore semaphore = new Semaphore(2,2);
public void Process(string documentId)
{
  semaphore.WaitOne();
  Document document;
  try
  {
    document = ReadDocumentFromSourceStore(documentId);
  }
  finally
  {
    semaphore.Release();
  }

  var translatedDocument = TranslateDocument(document, Language.English);

  SaveDocumentToDestinationStore(translatedDocument);
}
```

Note: *Limiting access to `ReadDocumentFromSourceStore` is a concern that can be extracted to its own decorator instead of being tangled with the processing code. In this article however, my focus is not on coding practices.*

For more information about the Semaphore class, take a look at this reference: https://docs.microsoft.com/en-us/dotnet/api/system.threading.semaphore?view=netframework-4.7

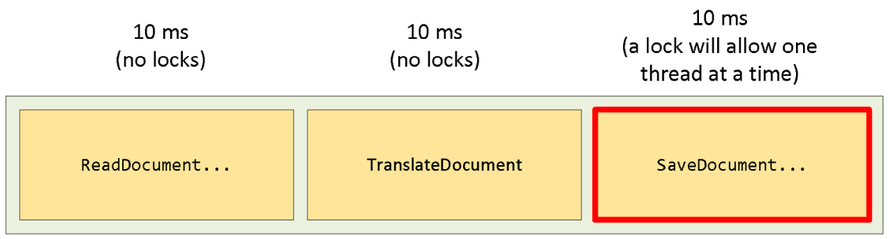
For more information about locks in C#, take a look at this reference: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/lock-statement

`SaveDocumentToDestinationStore` will probably also not be able to handle being called by eight threads in parallel. We can use a lock or a semaphore to also limit the number of threads that can access this method.

Ever changing execution times

When we use a lock or a semaphore, some threads will be blocked waiting for their turn to enter the protected section of the code, e.g. to call `SaveDocumentToDestinationStore`. In many cases, this makes a lot of sense.

Consider the following example:

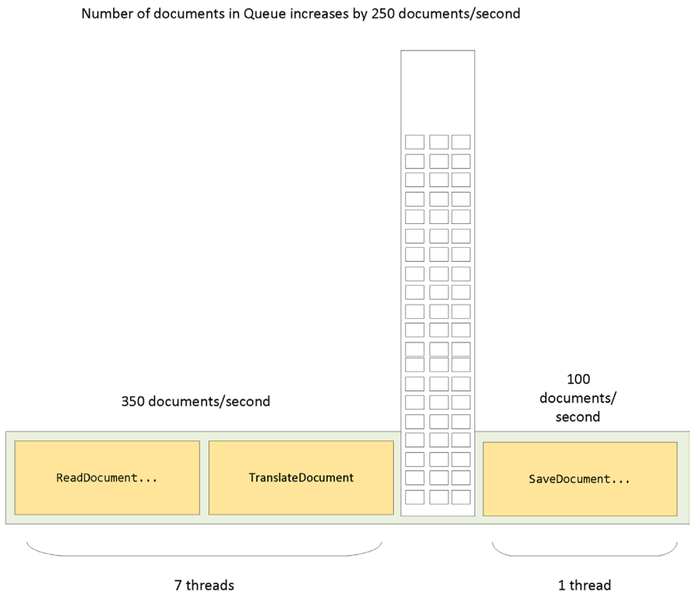


In this example, each of the three operations takes 10 millisecond to complete. However, the `SaveDocumentToDestinationStore` method is protected with a lock to allow only one thread to call it (for reasons discussed in the previous section). Note that in this example, we assume that `ReadDocumentFromSourceStore` can be called from multiple threads without an issue.

Without the lock, each thread will process a document in 30 milliseconds (assuming all operations can be parallelized without an issue). Now, with the lock, this is no longer true. A thread would take 30 milliseconds plus the time it has to wait to obtain the lock (its turn to execute `SaveDocumentToDestinationStore`).

Instead of having the threads waiting and doing nothing, can we make them grab other document ids, read the documents and translate them?

The following figure illustrates this:



In this example, seven threads read and translate documents.

Each one of these threads will take one document id, call `ReadDocumentFromSourceStore` and `TranslateDocument`, then put the result of the translation into a queue and go back to process (read and translate) another document, etc. A dedicated thread will take documents from the queue and call `SaveDocumentToDestinationStore` and then go fetch another document from the queue, etc.

Because there are seven threads working on the left of the queue, we can process about 7 threads * (1000 millisecond / (10 millisecond + 10 millisecond)) = 350 documents per second. The single thread on the right of the queue will be processing 1000 milliseconds / 10 milliseconds = 100 documents per second. This means that the number of items in the queue will increase by 250 documents per second. After all documents have been read and translated, the number of documents in the queue will start to decrease by 100 documents per second, i.e., the rate at which we save documents.

So even if we partially process (read and translate) documents faster than 100 documents per second, these partially processed documents would still need to wait (in the queue) for their turn to be saved and we are still going to have an overall throughput of 100 documents per second.

In summary, we gain nothing by making the threads process other documents, instead of waiting.

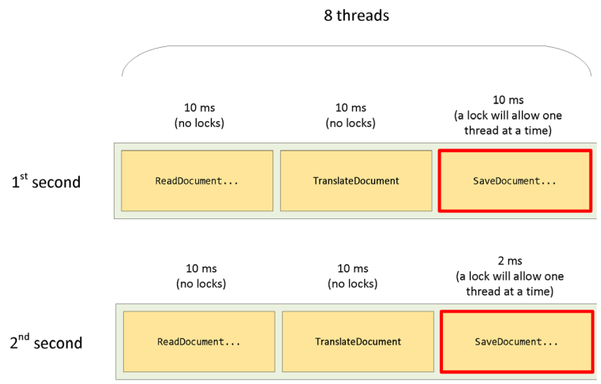
The conclusion we reached above is true because we assumed that the time it takes to save a document is constant (e.g. 10 milliseconds). However, in many cases, the time it takes to save a document changes frequently.

Consider the case where the document is saved to a disk on a remote machine. The connection to the remote machine might become better or worse from time to time and therefore the time it takes to save a document might change frequently.

To analyze the effect of this let's assume that the `SaveDocumentToDestinationStore` method takes 10 milliseconds to complete half of the time and 2 milliseconds the other half (for example, as a result of low network load).

Let's consider the case where we process documents for the duration of two seconds. In the 1st second, `SaveDocumentToDestinationStore` takes 10 milliseconds to complete. In the 2nd second, it takes 2 milliseconds.

Let's first see what happens if there is no queue, i.e., when threads simply block to wait for their turn to call `SaveDocumentToDestinationStore`.

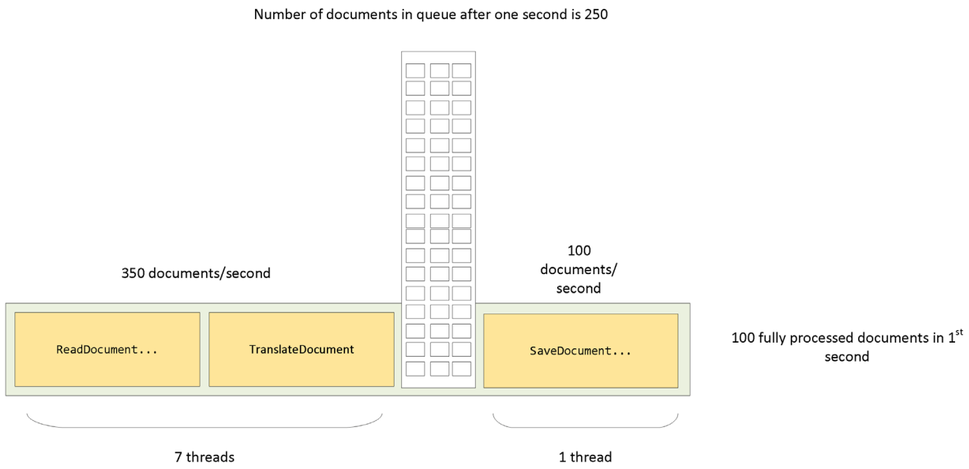


In the 1st second, we process 100 documents per second (as fast we can save documents) and therefore we process 100 documents.

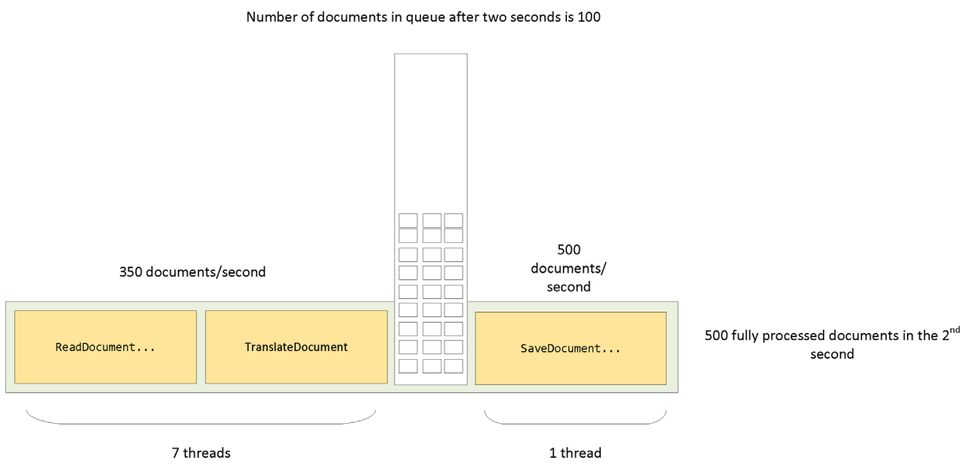
In the 2nd second, saving the document is no longer the bottleneck. In this second, each document will take 22 milliseconds + average waiting time for the lock. I am going to ignore this waiting time and say that we process 8 threads * 1000 milliseconds / 22 milliseconds ~= 363 documents (if we haven't ignored the waiting time, this would be less).

So, in the interval of two seconds, we process about 463 documents.

Let's now consider the case where we have a queue.



In the 1st second, we complete 100 documents. However, 250 (350 – 100) additional documents would have been read and translated and sitting in the queue.



In the 2nd second, the threads on the left can enqueue 350 additional documents. This means that a total of 600 documents are available for the saving thread to process. The thread on the right will process 500 documents in the 2nd second (1000 / 2).

This gives us a total of 600 documents processed in the whole interval of two seconds.

With this approach, we have processed at least 600 – 463 = 137 more documents in the interval of two seconds.

The Producer-Consumer pattern

In the previous section, I showcased two ways to process documents in parallel.

Such processing has three distinct stages: reading a document, translating it, and saving it.

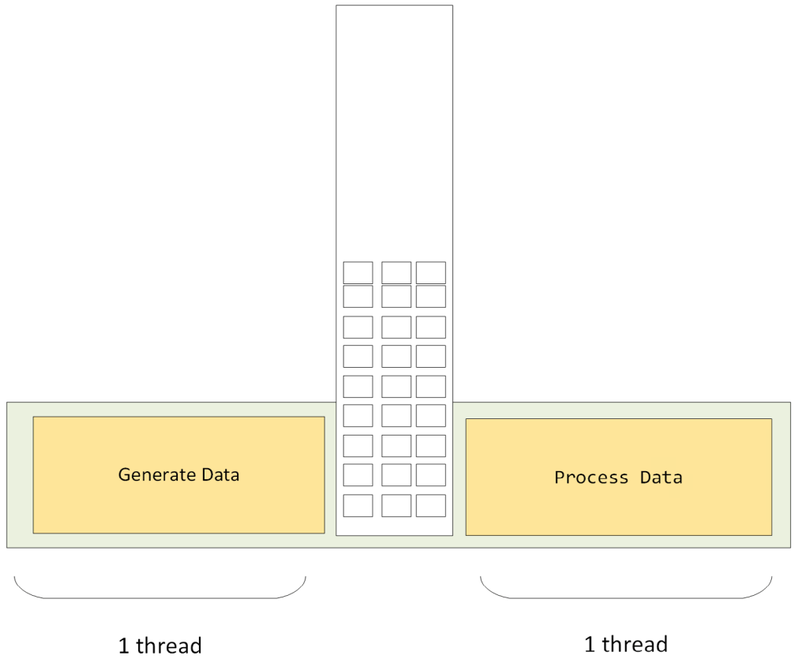
In the first way (the one without any queues), we have multiple threads and each will process a document through all three stages.

In the second way, each stage (or a group of stages) has dedicated threads running it.
In the last example, for instance, we had seven threads reading and translating documents, and a single thread saving documents.

This is basically the **producer-consumer pattern**.

In the example, we have seven producers that produce translated documents and we have a single consumer that consumes these documents by saving them. In this section, I will talk about this pattern in more details and show how it can be implemented in .NET.

When we apply the producer-consumer pattern in its simplest form, we will have a single thread generating data and enqueuing them into some queue, and another thread dequeuing the data from the queue and processing them.



The queue should be thread-safe. It is possible that one thread is enqueuing an item, while another one is dequeuing another item.

Sometimes, the producer can produce data faster than the consumer can process them. In this case, it makes sense to have an upper bound on the number of items the queue can store. If the queue is full, the producer will have to stop producing more data until the queue becomes non-full again.

Support for the Producer-Consumer pattern in the .NET framework

The easiest way to implement the producer-consumer pattern in .NET is to use the BlockingCollection<T> class.

The following code shows how we can implement the document processing example using this class:

```
public void ProcessDocumentsUsingProducerConsumerPattern()
{
  string[] documentIds = GetDocumentIdsToProcess();

  BlockingCollection<string> inputQueue = CreateInputQueue(documentIds);

  BlockingCollection<Document> queue = new BlockingCollection<Document>(500);

  var consumer = Task.Run(() =>
  {
    foreach (var translatedDocument in queue.GetConsumingEnumerable())
    {
      SaveDocumentToDestinationStore(translatedDocument);
    }
  });

  var producers = Enumerable.Range(0, 7)
    .Select(_ => Task.Run(() =>
    {
      foreach (var documentId in inputQueue.GetConsumingEnumerable())
      {
        var document = ReadAndTranslateDocument(documentId);
          queue.Add(document);
      }
    }))
    .ToArray();

  Task.WaitAll(producers);

  queue.CompleteAdding();

  consumer.Wait();
}

private BlockingCollection<string> CreateInputQueue(string[] documentIds)
{
  var inputQueue = new BlockingCollection<string>();

  foreach (var id in documentIds)
    inputQueue.Add(id);

  inputQueue.CompleteAdding();

  return inputQueue;
}
```

In this example, we first call `GetDocumentIdsToProcess` to get the list of document ids that need to be processed. Then all the document ids are added into a new `BlockingCollection` object (in the `inputQueue` variable) by using the `CreateInputQueue` method. For now, ignore the fact that we use a `BlockingCollection` to store the document ids, we're just using it here as a thread-safe collection with convenient API. I will explain this later.

We then create a `BlockingCollection` object specifying a maximum queue size of 500 items. This object will be used as the queue between the producers and the consumer. Having a limit on the queue size will cause a producer to block when it tries to add an item if the queue is full.

We then create a task (via `Task.Run`) to consume documents from the queue.

We use the GetConsumingEnumerable method to obtain an `IEnumerable<Document>` that can be used to loop through the produced documents as soon as they are available. If the queue is empty, the call to `IEnumerable<Document>`.MoveNext would block to cause the loop to pause. In the loop body, we simply invoke `SaveDocumentToDestinationStore` to save the translated documents.

Next, we create seven tasks that will produce the translated documents.

Each producer will call `GetConsumingEnumerable` on the document ids `BlockingCollection` (in the `inputQueue` variable) to get some document ids to process. For each document id, a producer will read and translate the document and then add it to the queue via the `Add` method.

Please note that the `BlockingCollection` class handles multiple threads reading items from it correctly, i.e., no single item will be read by two threads.

In case you are wondering why we do we have a `BlockingCollection` for the input queue, it is simply convenient as I explained just now. Had we stored the document ids in an array or a list, we would have to manage how the data is given correctly to the producer threads, ourselves.

Next, we wait for all producer tasks to complete via the Task.WaitAll method. Then, we invoke the CompleteAdding method on the `BlockingCollection` to mark the collection as completed.

When this happens, the consumer loop (the foreach loop) will complete once all items are processed. In more details, when the queue becomes empty, the `IEnumerable<Document>` returned from `GetConsumingEnumerable` will terminate (`IEnumerable<Document>.MoveNext` will return false). Without calling `CompleteAdding`, `IEnumerable<Document>.MoveNext` will simply block to wait for new documents to be added to the queue.

Finally, we wait for the consumer task to complete to make sure the `ProcessDocumentsUsingProducerConsumerPattern` method does not return before all documents are fully processed.

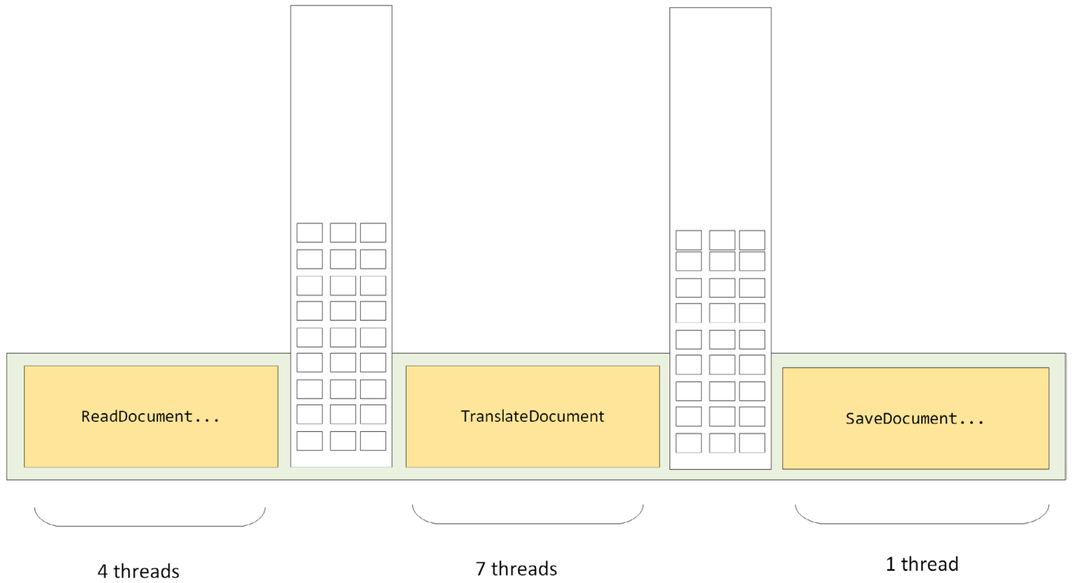
Please note that this is only one scenario where we can use the producer-consumer pattern. In other scenarios, there may not be a fixed-sized set of documents that we need to process and therefore it does not make sense to wait for the producer or consumer to finish. For example, we might want to have a never-ending process of reading new documents from an MSMQ queue and processing them.

The pipeline pattern

In our example, imagine that each stage of processing documents has its own degree of parallelism requirements and that its execution time changes frequently.

In this case, it makes sense to have two queues; one for read documents and one for translated documents.

The following figure illustrates this:



Basically, the pipeline pattern is a variant of the producer-consumer pattern.

In this pattern, some consumers are also producers. In this particular example, the translation process is both a consumer and a producer. It takes documents from the first queue, translates them, and then adds them to the second queue.

We can easily implement this pattern by using two `BlockingCollection` objects (and as explained in the previous example, we use another `BlockingCollection` to hold the input document ids for convenience):

```
public void ProcessDocumentsUsingPipelinePattern()
{
 string[] documentIds = GetDocumentIdsToProcess();
 BlockingCollection<string> inputQueue = CreateInputQueue(documentIds);
 BlockingCollection<Document> queue1 = new BlockingCollection<Document>(500);
 BlockingCollection<Document> queue2 = new BlockingCollection<Document>(500);
  var savingTask = Task.Run(() =>
  {
   foreach (var translatedDocument in queue2.GetConsumingEnumerable())
   {
    SaveDocumentToDestinationStore(translatedDocument);
   }
  });

  var translationTasks =
   Enumerable.Range(0, 7)
    .Select(_ =>
     Task.Run(() =>
      {
       foreach (var readDocument in queue1.GetConsumingEnumerable())
        {
         var translatedDocument =
          TranslateDocument(readDocument, Language.English);
           queue2.Add(translatedDocument);
        }
       }))
       .ToArray();

var readingTasks =
 Enumerable.Range(0, 4)
  .Select(_ =>
   Task.Run(() =>
    {
     foreach (var documentId in inputQueue.GetConsumingEnumerable())
     {
      var document = ReadDocumentFromSourceStore(documentId);
      queue1.Add(document);
     }
    }))
    .ToArray();

Task.WaitAll(readingTasks);

queue1.CompleteAdding();

  Task.WaitAll(translationTasks);

  queue2.CompleteAdding();

  savingTask.Wait();
}
```

A note about I/O operations and server applications

The read and save operations in the example I used in the article, are I/O operations. If we synchronously invoke an I/O operation, e.g. by using File.ReadAllBytes, the calling thread will block while the operation is running. When a thread blocks, it does not consume CPU cycles. And in desktop applications, having a few threads that block on I/O is not an issue.

In server applications however, e.g. WCF applications, the story is different.

If we have hundreds of concurrent requests of which processing requires accessing some I/O, it will be much better if we don't block on I/O. The reason is that I/O does not require a thread, and instead of waiting, the thread can go and process another request and therefore enhance throughput.

In the examples I used in the article, all I/O operations are done synchronously and therefore they are not suitable for server applications that require high throughput.

It also makes sense not to block the current thread when we need to wait for a producer-consumer queue to become non-empty (on the consumer side) and non-full (on the producer side).

The TPL Dataflow Library can be used to implement the producer-consumer and pipeline patterns. It has good support for asynchronous operations and can be used in server scenarios.

The Dataflow pattern

The TPL Dataflow Library mentioned above also supports another variant of the producer-consumer pattern called the Dataflow pattern.

The Dataflow pattern is different from the pipeline pattern in that the flow of data does not have to be linear. For example, you can have documents go to different queues based on some condition. Or you can send a document to two queues to have them translated to both English and Spanish.

In an upcoming article, I am going to discuss this pattern in more details. I will also show examples of how to better handle I/O operations in server applications.

Conclusion:

In this article, I have discussed the producer-consumer pattern and one variant of this pattern; the pipeline pattern.

I discussed the reasons why one would use these patterns instead of simple parallelism. One reason to use these patterns is that they give us more control over the degree of parallelism that each stage of processing has. It also enhances throughput when the execution time of some operations changes frequently.

I also provided examples of how to implement these patterns using the BlockingCollection class in .NET

Yacoub Massad

Author

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

Thanks to Damir Arh for reviewing this article.

Daniel Jimenez Garcia

WHAT'S NEW IN ASP.NET CORE 2.0

In this article, we will focus on ASP.NET Core 2.0 by taking a look at the most important new features and improvements over its previous versions. If you haven't done so yet, I would recommend combining this reading with the linked announcements. I hope you will enjoy it!

You can find the **companion source code** on github.

In August 2017, the ASP.NET team announced the public release of ASP.NET Core 2.0, released at the same time with .NET Core 2.0, Entity Framework Core 2.0 and the .NET Standard 2.0 specification.

This is great news for us developers as Microsoft's cross-platform offering keeps maturing, improving and receiving new features. The release of .NET Standard 2.0 has also considerably increased the number of APIs that are now available in .NET Core, with the framework now covering a higher number of the use cases needed by existing teams and/or projects.

EDITORIAL NOTE

If you want to stay updated with what's going on in the .NET space, make sure to read http://www.dotnetcurry.com/dotnet/1391/dotnet-developer-what-is-new

GETTING STARTED WITH ASP.NET CORE 2.0

If you haven't done so, download and install the latest version of .NET Core which also includes ASP.NET Core from the official dotnet site. If you use Visual Studio, make sure you also install the latest updates. Once you have done so, creating a new ASP.NET Core 2.0 is as simple as executing the `dotnet new` command using any of the web installed templates like `dotnet new mvc` or `dotnet new react`.

Since the list of templates has changed, simply execute `dotnet new` to get the full list:

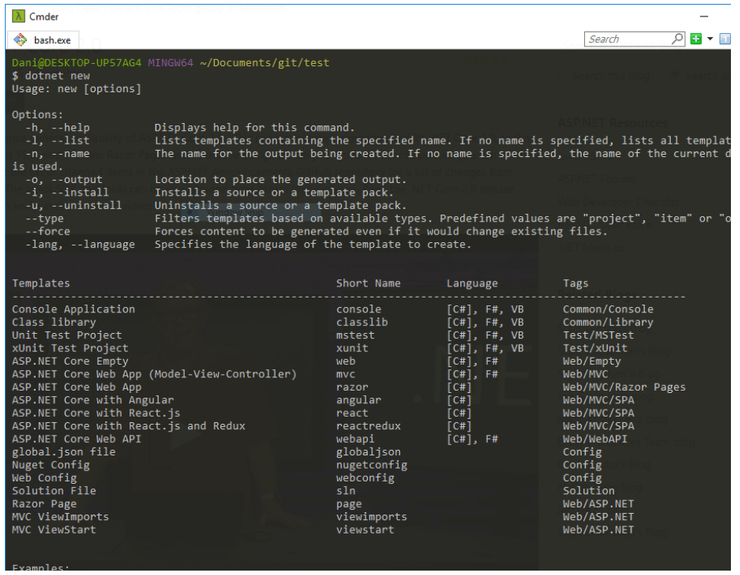


Figure 1, available project templates with dotnet new

Alternatively, you can start Visual Studio 2017 and execute File > New > Project then select the ASP .NET Core Web Application template in the dialog that appears:

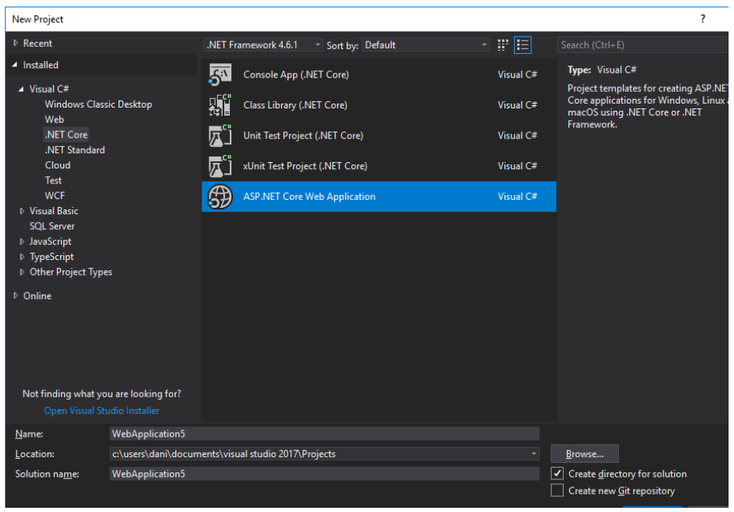


Figure 2, creating new ASP.NET Core project in Visual Studio

Once you do so, a second dialog will appear where you will be able to select which framework to target (by default the cross platform .NET Core is selected with version 2.0) and the specific ASP.NET Core template to use:

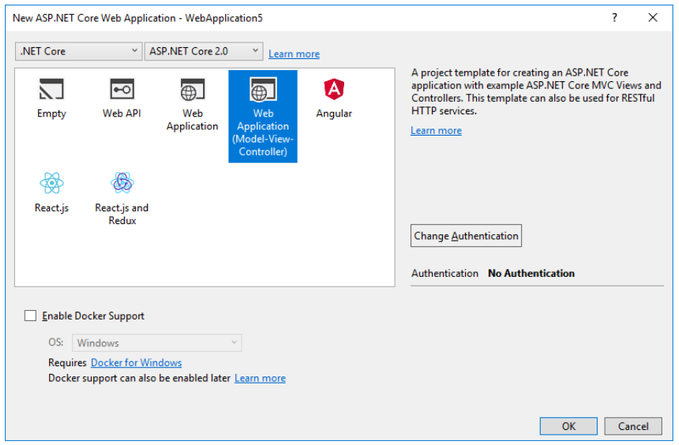


Figure 3, selecting the framework and template for the new ASP.NET Core project in Visual Studio

Upgrading existing ASP.NET Core 1.x projects

If you already have an ASP.NET Core 1.X project and would like to upgrade, Microsoft has published a handy migration guide.

I do not wish to repeat the guide here, so I will just quickly summarize the most important steps:

- Upgrade the target framework to either .NET Core 2.0 or the latest full .NET Framework (depending on whether you were targeting .NET Core or not before)
- Upgrade the package references to their 2.0 versions and take advantage of the new `Microsoft.AspNetCore.All` metapackage
- Update the Main method in Program.cs following the new host builder pattern (more later in the article)
- Update the Startup class, moving the Configuration and Logging setup to the Main method (more about this also later in the article).

Additional steps might be needed if you use for example EF Core or want to take advantage of some of the new features.

⚠ One little warning before we move on.

In previous versions of the framework you could create a global.json file on any folder and set the framework target version of that folder and sub-folders. If you have such a file in one of your existing projects or in any of its parent folders, make sure you update it to target the version 2.0

- If you forget to do so, once you upgrade visual studio to the latest 15.3 version, you will start getting the error *"The SDK 'Microsoft.NET.Sdk.Web' specified could not be found"* in visual studio when opening the project
- Andrew Lock's wrote in detail about this issue in his blog.

With that last piece of advice, let's move on to what's new in ASP.NET Core 2.0!

ASP.NET Core 2.0 - What's New

Razor pages

Razor pages is one of the biggest (if not THE biggest) new features introduced. However, you might be wondering how could that be, since we have been writing Razor *pages* for quite some years?

That's because by **Razor Pages**, we are now referring at a new MVVM (Model-View-ViewModel) paradigm that's available in ASP.NET Core projects.

Don't be afraid, this is not a replacement of the previous MVC paradigm, in fact it's frictionless to use both MVC and MVVM depending on the problem you are solving.

Look at it as another tool in your toolbox that will make your life easier when used in the right situations.

For example, have you ever looked at the HomeController created for the SPA (single page applications) templates like Angular or React and wondered what benefits is it bringing?

```
public class HomeController : Controller
{
  public IActionResult Index()
  {
    return View();
  }
  public IActionResult Error()
  {
    return View();
  }
}
```

Wouldn't it be great if all you had to do was create the View file without the need for this pure boilerplate controller?

This is just one of the examples where Razor pages and its MVVM paradigm might be a better approach than MVC. You will now have the flexibility to use one or the other for each specific use case!

I have upgraded the SPA Vue.js project I used in an earlier article to use ASP.NET Core 2.0 and Razor Pages. Check out the commit with the relevant changes!

Getting started with razor pages

Once you have an ASP.NET Core 2.0 project, Razor pages are enabled by default when using `services.AddMvc();` in your Startup.ConfigureServices method.

Let's give it a try, create a folder named **Pages** at the root of your project (same place where the Views folder is located) and then create inside a new file `HelloWorld.cshtml` with the following contents:

```
@page
<h1>Hello World!</h1>
```

```
<p>The server time is @DateTime.Now</p>
```

Apart from the `@page` directive, everything else is the same as any other Razor view you have seen so far.

Now build and run the project and navigate to `/HelloWorld`. Congratulations, you just rendered your first Razor page:

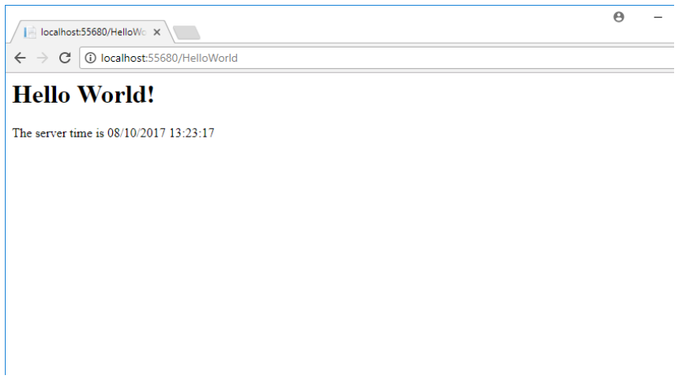


Figure 4, first simple Razor page being rendered

Of course, that is rather ugly, and you would like to have a layout and some structure like regular views do.

Since this is Razor, everything you know about Razor views can be applied to Razor pages. This is not just the syntax, for example the **layouts** and **special _ViewStart.cshtml** and **_ViewImports.cshtml** behave the same way.

Copy the `_Layout.cshtml` and `_ViewStart.cshtml` files located inside /Views/Shared into the /Pages folder. Now restart the project and navigate again to `/HelloWorld`, this time you will see how the default layout has also been applied to the Razor page:

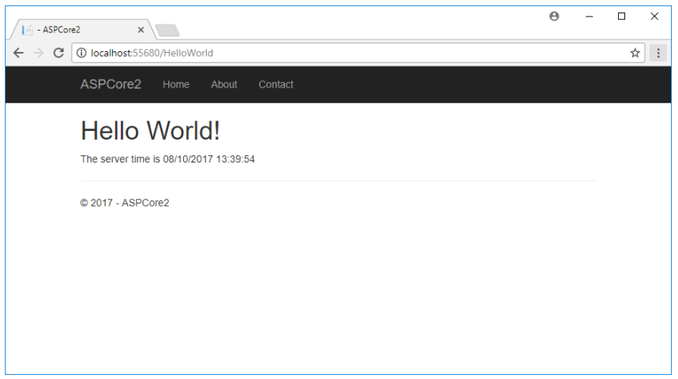


Figure 5, the simple Razor Page using the layout

You might be wondering what happens if your project doesn't have Views and only has Razor Pages? Or if you want to use a different layout for Pages than the one for Views?

Well you can just create a file named `_Layout.cshtml` in the Pages folder and define the layout for the Pages.

Note: Try that by copying the one located inside `/Views/Shared` and making some changes to it.

Before we start looking at Razor pages in more detail, I should mention there is a project template available at the CLI with `dotnet new razor` that will initialize the same as the typical MVC project, but using only razor views.

If you are still unsure about what Razor Pages are, generate 2 projects using `dotnet new razor` and `dotnet new mvc` and compare the Razor Pages against the Controller+Views.

Routing

By default, routes to Razor Pages are based on the name and location of the page files:

File name and location	Route
/Pages/Index.cshtml	/ or /Index
/Pages/HelloWorld.cshtml	/HelloWorld
/Pages/Foo/Bar/Index.cshtml	/Foo/Bar or /Foo/Bar/Index
/Pages/Foo/Bar/HelloWorld.cshtml	/Foo/Bar/HelloWorld

As you can see there is almost a 1:1 mapping between routes and folder/file names, except for the special `Index.cshtml` case.

Even though Razor Pages are designed with this simple mapping in mind, it supports many of the features you are now used to when defining your application routes like route parameters, constraints or even your own conventions..

For example, using the `@page` directive you can append additional segments and even parameters to the URLs we have seen before. Replace the contents of `Pages/HelloWorld.cshtml` with the following:

```
@page "foo/{id}"
<h1>Hello World!</h1>
<p>You have provided the id: @RouteData.Values["id"]</p>
```

You will now get a 404 if you navigate to `/HelloWorld`, but will see the page if you navigate to `/HelloWorld/foo/42`:

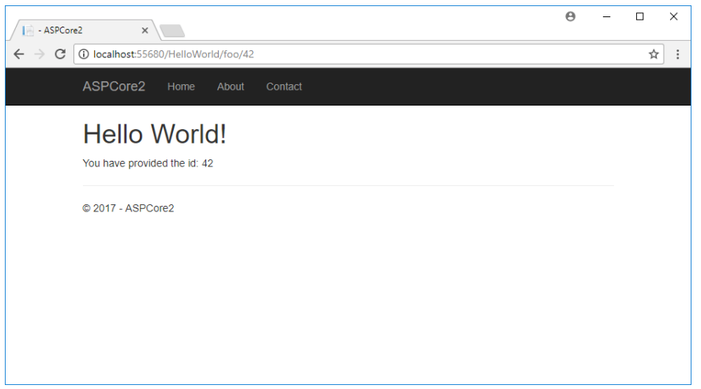


Figure 6, the Razor Page with the updated route

The entire range of route constraints are available when defining route parameters in Razor Pages.

Finally, you can update the default conventions or even add your own ones using the `AddRazorPagesOptions` in the `Startup.ConfigureServices` method.

For example, you can fully replace the automatically generated route with one of your own:

```
services.AddMvc()
  .AddRazorPagesOptions((opts) =>
  {
    opts.Conventions.AddPageRoute("/HelloWorld", "hello-world");
  });
```

The previous page will now be associated with the route `/hello-world` (notice how the override also affects the bits defined with the `@page` directive).

A more interesting example is a convention that replaces camel case names into kebab case, so `/FooBar/HelloWorld.cshtml` automatically becomes `/foo-bar/hello-world`:

```
services.AddMvc()
  .AddRazorPagesOptions((opts) =>
  {
    opts.Conventions.AddFolderRouteModelConvention("/", model =>
    {
      foreach (var selector in model.Selectors)
      {
        var attributeRouteModel = selector.AttributeRouteModel;
        attributeRouteModel.Template = ToKebabCase(attributeRouteModel.Template);
      }
    });

private string ToKebabCase(string s)
{
  return Regex.Replace(
    s,
    "([a-z])([A-Z])",
    "$1-$2",
    RegexOptions.Compiled)
    .ToLower();
}
```

This has the added benefit of playing well with customizations made in the `@page` directive, so our `/Hello-World.cshtml` will now be associated with `/hello-world/foo/{id}`:

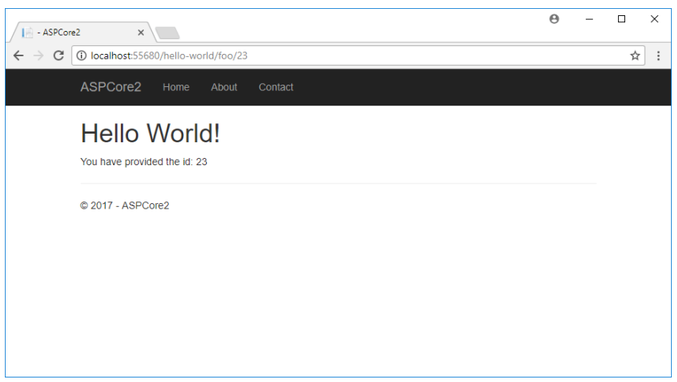


Figure 7, route updated through convention and @page directive

Page Models

When using Razor Pages, sooner or later you will need to encapsulate and move the logic out from the view. Within a traditional MVVM architecture you will be looking at moving this into the VM, while in the Razor Pages world, you will be creating a Page Model.

Let's revert the previous routing changes and get back to our simple page `/Pages/HelloWorld.cshtml` with the following contents:

```
@page
<h1>Hello World!</h1>
<p>The server time is @DateTime.Now</p>
```

Let's add a very simple page model for this page. Start by creating a new class named HelloWorld.cshtml.cs located in the same folder with the following contents:

```
public class HelloWorldModel: PageModel
{
  public DateTime ServerTime { get; set; }

  public void OnGet()
  {
    // This method is invoked when the page gets rendered in response to a GET
    request
    ServerTime = DateTime.Now;
  }
}
```

Then update the Page itself to read like:

```
@page
@model ASPCore2.Pages.HelloWorldModel
<h1>Hello World!</h1>
<p>The server time is @Model.ServerTime</p>
```

As you can see, we have just created our model inheriting from PageModel and we are using the `@model` directive in the page to link our Razor Page with its model. You can then use any of its properties as in `Model.ServerTime`.

- When you start creating multiple pages with corresponding models, you can avoid specifying the full namespace in the `@model` directive if instead, you add a `@namespace` directive to the `_ViewStart.cshtml` file like:

  ```
  @namespace ASPCore2.Pages
  ```

- Then you can simply use the model class name with the @model directive as in `@model HelloWorldModel`

It's also worth mentioning that the model file doesn't really have to be named like the page itself, since they are bound through the `@page` directive which is based on the model class name. However, if you follow the file naming convention, Visual Studio will recognize it and collapse the model inside the page:

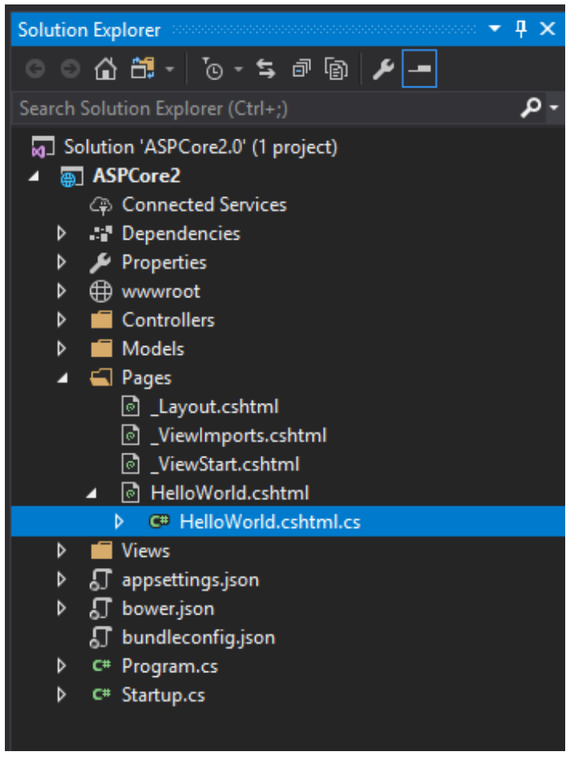


Figure 8, page and its model grouped together in Visual Studio

This is in fact the convention followed by the Razor Page file template in Visual Studio! Try it by right clicking the Pages folder, then select Add > New Item and finally select the Razor Page template.

Dependency Injection in Page Models

A Page Model would be of limited use if it couldn't communicate with the rest of the system. It is responsible not only for the data binding, but also for initiating the necessary calls to fetch/update your data in response to the different HTTP requests.

The good news is that they support dependency injection out of the box, all you need to do is to add a constructor receiving the desired dependencies.

For example, update the previous Page Model to read like:

```
public class HelloWorldModel: PageModel
{
  public HelloWorldModel(IHostingEnvironment env)
  {
    PhysicalPath = env.ContentRootPath;
  }

  public string PhysicalPath { get;}
  public DateTime ServerTime { get; set; }

  public void OnGet()
  {
    ServerTime = DateTime.Now;
  }
}
```

And now, we can update the view to show the physical location of our server:

```
<p>The server files are located at @Model.PhysicalPath</p>
```

Just like that, dependency injection works out of the box as you would expect in Razor Pages.

URL parameters

We saw earlier how it is possible to use the `@page` directive to define url parameters part of the route associated with the page. For example:

- When using `@page "{id}"` your page will be associated with the `/HelloWorld/{id}` URL, where the id parameter is required.
- You can define an optional parameter adding a '?' at the end as in `@page "{id?}"`, so now both /HelloWorld and /HelloWorld/42 would render your page.
- You can add any of the available constraints to the page URL parameters

Let's update our simple page with the directive `@page "{id:int}"` so an integer id is now part of the route.

The question now is **how do you read the value of the parameter in your page model?**

While you could manually parse through the `RouteData` dictionary, a cleaner alternative is to add a parameter to the `OnGet` method:

```
public void OnGet(int id)
{
  Id = id;
  ServerTime = DateTime.Now;
}
```

Finally, a similar approach would be using the `[BindProperty]` attribute with the Id property:

```
[BindProperty(SupportsGet = true)]
public int Id { get; set; }
public DateTime ServerTime { get; set; }

public void OnGet()
{
  ServerTime = DateTime.Now;
}
```

One last bit before we move on.

How does one generate a URL to a certain page, including any required parameters?

There are a couple of ways, both of which are very similar to how you generate a URL to a certain controller action:

- If you need to generate the URL itself, you can use the UrlHelper.Page method, for example in a razor page you can do:

```
@Url.Page("HelloWorld", new {id = 100})
```

- If you want to generate a full link tag pointing towards a specific razor page, then you would use the link tag helper setting the asp-page attribute and any required route attributes:

```
<a asp-page="/HelloWorld" asp-route-id="100">My Razor Page</a>
```

I hope all we have seen so far regarding Razor Pages is intuitive enough for you!

You might be wondering about the `OnGet` method of the Page Model though! Don't worry we will see how pages can handle different types of requests in the next section

Form actions

So far, we have seen page models with an `OnGet` method, which as you might have guessed, is executed in response to an HTTP GET request.

But what if your page includes a form and you want the Page Model to handle additional requests like a POST?

The framework will look for a method `OnPost` (or OnPostAsync, the Async suffix is optional) in the Page Model class. The only restriction is that it should return an `IActionResult` (or `Task<IActionResult>` when using async methods)

Let's see a very simple example using the well-known problem of the Todo list.

Start by adding a simple TodoModel to the models folder:

```
public class TodoModel
{
  [Required]
  public string Title { get; set; }
  [Required]
  public string Description { get; set; }
  public bool Completed { get; set; }
}
```

Let's also add a static collection to the model itself, which will serve as a very simple in-memory repository.

Note: *this is just to keep the example focused on the action handler without distracting the reader with repositories, entity framework or dependency injection. Please don't do this in real applications!*

```
public static ConcurrentBag<TodoModel> Todos = new ConcurrentBag<TodoModel>();
```

Let's now create a new folder named Todo inside the Pages folder. Create a new Razor Page named `Create.cshtml` together with its model `Create.cshtml.cs`.

Next add an `OnGet` method where we initialize a new instance of the TodoModel:

```
[BindProperty]
public TodoModel Todo { get; set; }

public void OnGet()
{
  Todo = new TodoModel { Title = "What needs to be done?" };
}
```

Now update the page itself so it renders a form where users can enter the values of the new Todo:

```
@page
@model CreateModel
<h1>Add new Todo</h1>
<form method="post">
    <div asp-validation-summary="All"></div>
    <div class="form-group">
        <label asp-for="Todo.Title"></label>
        <div>
            <input class="form-control" asp-for="Todo.Title" />
            <span asp-validation-for="Todo.Title"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="Todo.Description"></label>
        <div>
            <input class="form-control" asp-for="Todo.Description" />
            <span asp-validation-for="Todo.Description"></span>
        </div>
    </div>

    <button class="btn bg-primary" type="submit">Add</button>
</form>
```

Finally, let's add the `OnPost` method that will handle the POST request, adding the new Todo to the in-memory list of Todos and redirecting to the Index page:

```
public IActionResult OnPost()
{
  if (!ModelState.IsValid)
  {
    return Page();
  }

  TodoModel.Todos.Add(Todo);

  return RedirectToPage("./Index");
}
```

Simple, right?

It is not much different from your typical controller actions, except for the different method naming convention and the fact that the posted data is binded to a property of the page model class which has the `[BindProperty]` attribute.

Let's complete our simple example by implementing the Index page. Update its Page Model so it gets the list of Todos from the in-memory repository:

```
public IEnumerable<TodoModel> Todos { get; set; }
public void OnGet()
{
  Todos = TodoModel.Todos.ToList();
}
```

Finally update its page so it renders each of the Todos and includes a link to the create page:

```
@page
@model IndexModel
<h1>Todo list</h1>
<ul class="list-unstyled">
  @foreach (var todo in Model.Todos)
```

```
  {
    <li>
      <p>@todo.Title - @todo.Description</p>
    </li>
  }
</ul>
<a asp-page="./Create">Add another</a>
```

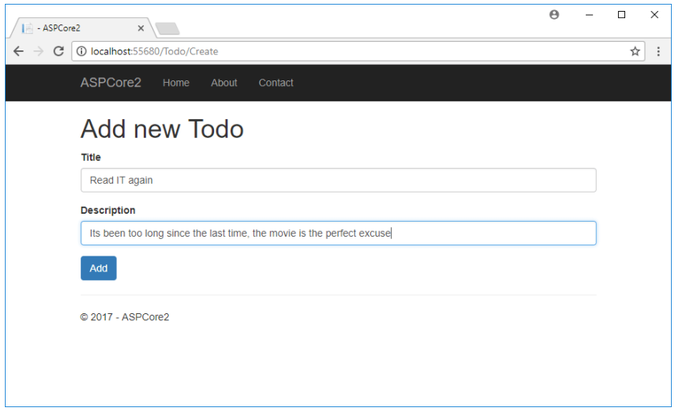


Figure 9, create todo page

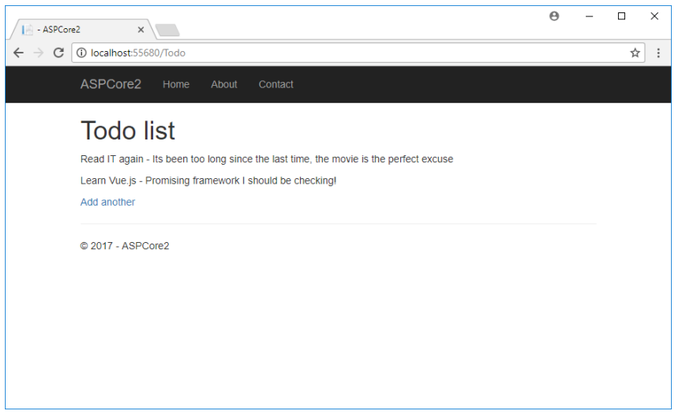


Figure 10, the page rendering a list of todos

This is where our overview of Razor Pages finishes!

I hope you agree it is very straightforward to get started with them and get used to the MVVM pattern they propose.

That doesn't mean they should be used everywhere (for example, I am not convinced about CRUD pages) but I can see how a combination of Razor Pages with GET-only pages and traditional controllers for APIs called by clients would be useful (particularly for SPAs).

If you want an even deeper look, I will recommend you to start from the official Microsoft documentation about Razor Pages.

Program and Startup classes

ASP.NET Core 2 simplifies the Program.cs once again, by streamlining the WebHostBuilder as part of its template.

Compare the default template you got when creating a 1.X project:

```
public class Program
{
  public static void Main(string[] args)
  {
    var host = new WebHostBuilder()
    .UseKestrel()
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseIISIntegration()
    .UseStartup<Startup>()
    .UseApplicationInsights()
    .Build();

  host.Run();
  }
}
```

With the one you now get in a 2.0 project:

```
public class Program
{
  public static void Main(string[] args)
  {
    BuildWebHost(args).Run();
  }

  public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
      .UseStartup<Startup>()
      .Build();
}
```

It might not seem like a huge difference but there are a few details worth describing:

- The most obvious one is that now it is recommended to separate the `Main` method that runs the application from the `BuildWebHost` which creates and configures an `IWebHost` but does not run it. And why would you care? This allows for tooling like scaffolding or EF migrations to inspect your IWebHost without running it

- The second is that instead of creating a `WebHostBuilder` and configure it, you can start from the utility `WebHost` class which provides a handy `CreateDefaultBuilder` method. This is roughly equivalent to the previous methods except for:

    - It adds the default configuration (appsettings.json, appsettings.{env}.json, user secrets in dev environment, environment variables and command line args). More about it in the next section!

- It adds the default logging (using settings from the logging configuration section it adds console and debug logging) More about it on the next section!
- It doesn't need to manually add application insights, since it now gets automatically started by means of the new `IHostingStartup` interface. More on it later!

You could check the actual definition of `CreateDefaultBuilder` in the aspnet MetaPackages repository, if you open the source code of the WebHost class.

You will see it looks very similar to the 1.X `Main` method, manually creating a `WebHostBuilder` and using the extension methods to configure it.

The fact that Configuration and Logging are now core services added in the `Program` class by the WebHostBuilder, means they no longer must be configured in the Startup class.

That way the startup class has also been simplified when compared to the default 1.X version:

- The configuration is simply injected in the constructor, since it has already been configured by the `WebHostBuilder` and is ready to be injected in any class that needs it.

- The Configure method no longer needs to receive an `ILoggerFactory` parameter and configure the logging, since logging has already been configured by the `WebHostBuilder.`

The default Startup class now looks like:

```
public class Startup
{
  public Startup(IConfiguration configuration)
  {
    Configuration = configuration;
  }

  public IConfiguration Configuration { get; }

  // This method gets called by the runtime.
  // Use this method to add services to the container.
  public void ConfigureServices(IServiceCollection services)
  {
    services.AddMvc();
  }

  // This method gets called by the runtime.
  // Use this method to configure the HTTP request pipeline.
  public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    if (env.IsDevelopment())
    {
      app.UseDeveloperExceptionPage();
      app.UseBrowserLink();
    }
    else
    {
      app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseMvc(routes =>
    {
      routes.MapRoute(
        name: "default",
       template: "{controller=Home}/{action=Index}/{id?}");
    });
  }
}
```

You can read more about hosting and startup of 2.0 applications in the official Microsoft documentation.

Logging and Configuration

Both Logging and Configuration are now configured in the Program class by the `WebHostBuilder` and available as services that can be injected, so:

- The interface `IConfiguration` is now available for dependency injection and can be injected anywhere (whereas before you would have to use the `IOptions<T>` or manually add the `IConfiguration` to the services)

- The interfaces `ILogging`, `ILoggingFactory` and `IHostingEnvironment` are now available for dependency injection from earlier, which might come handy for some initialization code.

As we have seen in the previous section, the new `WebHost.CreateDefaultBuilder` used in the Main class will add Configuration and Logging using their default strategies. But of course, you can keep using the methods available in IWebHostBuilder to override or expand these defaults:

```
public static IWebHost BuildWebHost(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureAppConfiguration((context, config) => {
      // If you want to override default sources, uncomment the next line
      //config.Sources.Clear();

      // Add new sources on top of the default config
      config.AddJsonFile("foo.json");
    })
    .ConfigureLogging((context, logging) => {
      // If you want to override default sources, uncomment the next line
      //logging.ClearProviders();

      // Add new logger providers on top of the default ones
      logging.AddEventSourceLogger();
    })
    .Build();
```

Logging Filtering

Another very interesting improvement to the logging framework is that now in the configuration, you can simply define that the logging level is applicable to all or specific loggers and categories. There is an application wide `loggingFactory.AddConfiguration`, conveniently called by the default WebHostBuilder.

Note: In earlier releases you would find yourself either passing the configuration to each logger registration or using the configuration within the `loggerFactory.WithFilters` helper.

For example, given the default logger (debug and console) added to a project, you can now update your configuration as follows in order to filter which event gets logged to each provider (example is taken from the official Microsoft docs about Logging):

```
{
  “Logging”: {
    “IncludeScopes”: false,
    “LogLevel”: {
      “Default”: “Debug”,
      “System”: “Information”,
      “Microsoft”: “Information”
    },
    “Debug”: {
      “LogLevel”: {
        “Default”: “Information”,
        “Microsoft.AspNetCore.Mvc.Internal”: “Error”,
        “Microsoft.AspNetCore.Hosting.Internal”: “Error”
      }
    }
  }
}
```

The top-level `LogLevel` section applies to any logger while the `LogLevel` nested under Debug applies only to the Debug logger. (For a list of built-in providers check the Microsoft docs).

Since the logging configuration is now applied at the framework level, it will filter messages for any logger you add to your application. This means if you create your own logger or add a 3rd party one, you can use the configuration to define the filtering rules for that logger as well with zero coding effort.

Let’s say you create your own logger, like the following (useless) logger:

```
public class FooLoggerProvider: ILoggerProvider
{
  public ILogger CreateLogger(string categoryName)
  {
    return new FooLogger(categoryName);
  }

  public void Dispose()
  {
  }
}

public class FooLogger: ILogger
{
  private readonly string _categoryName;

  public FooLogger(string categoryName)
  {
    _categoryName = categoryName;
  }

  public void Log<TState>(LogLevel logLevel, EventId eventId, TState state,
  Exception exception, Func<TState, Exception, string> formatter)
  {
    var message = formatter(state, exception);
    System.Diagnostics.Debug.WriteLine(message, _categoryName);
  }

  public IDisposable BeginScope<TState>(TState state){
    return null;
  }

  public bool IsEnabled(LogLevel logLevel){
    return true;
  }
}
```

And you register it on your Main class:

```
public static IWebHost BuildWebHost(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .ConfigureLogging((context, logging) =>
    logging.AddProvider(new FooLoggerProvider()))
    .Build();
```

You can then update your configuration to define specific filters that apply to that logger only:

```
{
  “Logging”: {
    “IncludeScopes”: false,
    “LogLevel”: {
      “Default”: “Debug”,
      “System”: “Information”,
      “Microsoft”: “Information”
    },
    “ASPCore2.FooLoggerProvider”: {
      “LogLevel”: {
        “Default”: “Information”,
        “Microsoft.AspNetCore.Mvc.Internal”: “Error”,
        “Microsoft.AspNetCore.Hosting.Internal”: “Error”
      }
    }
  }
}
```

Notice how the `LogLevel` section is nested under a section with the full type name of the logger provider. This might get a bit annoying, however you can add the `[ProviderAlias(“Foo”)]` attribute to the LoggerProvider and this way define it using the alias instead of the full type name:

```
{
  “Logging”: {
    “IncludeScopes”: false,
    “LogLevel”: {
      “Default”: “Debug”,
      “System”: “Information”,
      “Microsoft”: “Information”
    },
    “Foo”: {
      “LogLevel”: {
        “Default”: “Information”,
        “Microsoft.AspNetCore.Mvc.Internal”: “Error”,
        “Microsoft.AspNetCore.Hosting.Internal”: “Error”
      }
    }
  }
}
```

For more information about logging using the latest 2.0 features check the official Microsoft docs.

IHostingService

At first glance, `IHostingService` is a new and simple interface that only defines a pair of Start and Stop methods:

```
public class SampleHostedService: IHostedService
{
  public Task StartAsync(CancellationToken cancellationToken)
  {
    throw new NotImplementedException();
  }

  public Task StopAsync(CancellationToken cancellationToken)
  {
     throw new NotImplementedException();
  }
}
```

The power comes from the fact that if you register this class as a service in your application, then the framework will call the Start method during application startup and the Stop method during application shutdown.

This is very useful if you intend to run some background process independently of your main application, like the periodical refresh/upload/download of data or the running of certain processes in the background.

Let's update that simple process to basically just periodically add a logging (Warning, very simplistic code for the purposes of the article!):

```
private readonly ILogger<SampleHostedService> logger;
public SampleHostedService(ILogger<SampleHostedService> logger)
{
  this.logger = logger;
}

public Task StartAsync(CancellationToken cancellationToken)
{
  logger.LogInformation("Hosted service starting");

  return Task.Factory.StartNew(async () =>
  {
    while (!cancellationToken.IsCancellationRequested)
    {
       logger.LogInformation("Hosted service executing - {0}", DateTime.Now);
       try
       {
         await Task.Delay(TimeSpan.FromSeconds(10), cancellationToken);
       }
       catch (OperationCanceledException) { }
     }
  }, cancellationToken);
}

public Task StopAsync(CancellationToken cancellationToken)
{
  logger.LogInformation("Hosted service stopping");
  return Task.CompletedTask;
}
```

Update the startup class to register `SampleHostedService` as an `IHostedService`:

```
services.AddSingleton<IHostedService, SampleHostedService>();
```

Now let's start the application and we can see how our service is started as part of the application and keeps adding an entry every five seconds until the application is stopped (Try to use kestrel when starting the app instead of IISExpress or there will be no graceful shutdown, so StopAsync won't be called):

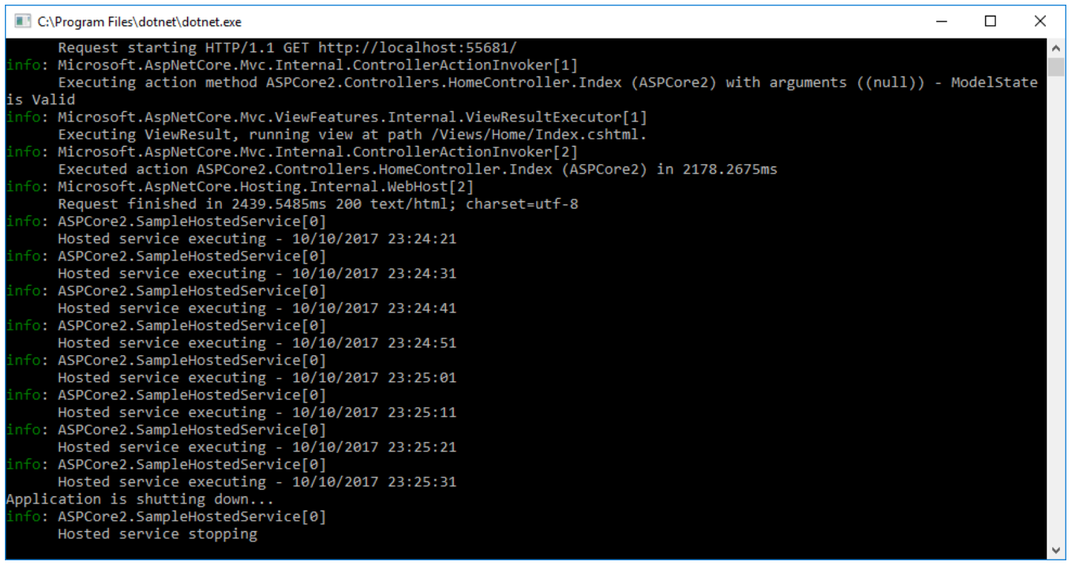


Figure 11, the simple IHostedService implementation in action

I am sure you can come up with several ways to use this new feature, although many of those scenarios would probably benefit by their own independent service anyway. Still, this is a welcome addition to the framework that enables scenarios which earlier *always* required an independent service.

Would you like to read more about it? Check these articles by Steve Gordon and Maarten Balliauw. (Especially if you want to see how to properly handle the cancellation tokens and background tasks instead of my very simple code!)

IHostingStartup

This new feature looks similar to `IHostedService` but is quite a different beast. While `IHostedService` is intended for you to use it as part of your application code if it would help towards solving some scenario; `IHostingStartup` is geared towards tooling.

The interface defines a single method:

```
public class SampleHostingStartup: IHostingStartup {
  public void Configure(IWebHostBuilder builder) {
    throw new NotImplementedException();
  }
}
```

This means you can use any of the IWebHostBuilder methods (and extension methods) to add or update additional behaviours and features. For example, I could add here the custom logging provider added in one of the earlier sections:

```
[assembly: HostingStartup(typeof(ASPCore2.SampleHostingStartup))]
namespace ASPCore2
{
  public class SampleHostingStartup : IHostingStartup
  {
    public void Configure(IWebHostBuilder builder)
    {
      builder.ConfigureLogging((context, logging) =>
      logging.AddProvider(new FooLoggerProvider()));
    }
  }
}
```

And why would you do that?

It seems quite a bit of indirection instead of simply configuring the logging provider in the Startup function…

Well, the contract is that the framework will discover all the `[HostingStartup]` **assembly** attributes that point to a concrete `IHostingStartup` implementation, and call them while building the `WebHost` in your Main class. And now the interesting bit, it will look for IHostingStartup implementations in (check the source code):

- Your main assembly (the one with the Main method)
- Any additional assemblies deployed within the application which are part of the semi-colon separated list found in the Configuration with key `hostingStartupAssemblies`.

You can now drop any assembly in your application folder, make sure the assembly name is part of the comma-separated list hostingStartupAssemblies configuration entry (For example, setting the ASPNETCORE_HOSTINGSTARTUPASSEMBLIES environment variables) and the method will be called during startup, allowing you to add extra services, configuration, loggers, etc

- And an even more obscure setting based on the DOTNET_ADDITIONAL_DEPS environment variable would include extra assemblies, check Gergely's article linked at the end of the section.

It's quite an interesting feature although specially geared towards tooling like IIS-Express or Application Insights (more on the next section), rather than a general-purpose plugin system. Hence the by-design explicit list of assemblies to scan for the attribute rather than wide scan, as per David Fowler's comment.

If you are interested in knowing more, then check this article from Gergely Kalapos.

Other Features

That's not everything that has changed on ASP.NET Core 2.0! There are a few other changes worth reading about.

Razor compilation on publish by default

Once you upgrade or start a new project using ASP.NET Core 2.0, Razor compilation on publish will be enabled by default.

That means when you package and publish your application, all the Razor Views/Pages will be compiled and deployed as part of your application assembly rather than separated cshtml that have to be compiled on demand.

This results in quicker startup time and faster initial responses!

- Although if you are targeting the full .NET framework, then you still need to reference a explicit NuGet package as per the official Microsoft docs.

If for some reason you want to turn off this feature, you can do so on your csproj file:

```
<PropertyGroup>
  <TargetFramework>netcoreapp2.0</TargetFramework>
  <MvcRazorCompileOnPublish>false</MvcRazorCompileOnPublish>
</PropertyGroup>
```

The Runtime Package Store and the AspNetCore.All metapackage

Microsoft has created a new metapackage named Microsoft.AspNetCore.All which includes references to all the other ASP.NET Core assemblies:

- The ASP.NET Core packages.
- The Entity Framework Core packages.
- Any additional dependencies needed either of those

The immediate benefit is that you need to maintain fewer references in your csproj and will have an easier time upgrading to a new baseline version (for example, from 2.0 to a future 2.X version).

In addition to that, by using the metapackage, you are also implicitly using the new .NET Core Runtime Package Store.

- The ASP.NET framework assemblies are already installed in the store with their native images ready. Consider it the GAC (Global Assembly Cache) of the .NET Core world.

- Your application package size is smaller since it doesn't need to include the assemblies in the Runtime Package Store (Unless you explicitly publish for self-contained deployment)

- Application startup can be improved when the Runtime Package Store already contains the native images of required assemblies.

TagHelperComponents

Did you use the MiniProfiler in any of your MVC applications?

This is a tool that collects profiling data from your server-side app and would include such information as

part of the rendered page so you could easily see it while running your app in development. However, it required you to add the required JS/CSS as part of your layout by manually adding `@MiniProfiler.RenderIncludes()` before closing the body tag.

This is one of the scenarios where the TagHelperComponents can be of help!

These are services registered as singletons, that will be called when rendering the app and will let you modify the contents of some tags.

Out of the box, the framework calls for both the **head** and **body** tags any TagHelperComponent that has been registered, giving you a chance of modifying the contents of those tags.

For example, a simplistic implementation of a TagHelperComponent that would add the CSS/JS files needed by miniprofiler could look like:

```
public class SampleTagHelperComponent: TagHelperComponent
{
  public override int Order => 100;

  public override void Process(TagHelperContext context, TagHelperOutput output)
  {
    if (string.Equals(context.TagName, “head”, StringComparison.OrdinalIgnoreCase))
    {
      output.PostContent.AppendHtml(“<link rel=’stylesheet’ type=’text/css’
      href=’miniprofiler.css’></link>”);
    }
    if (string.Equals(context.TagName, “body”, StringComparison.OrdinalIgnoreCase))
    {
      output.PostContent.AppendHtml(“<script src=’miniprofiler.js’></script>”);
    }
  }
}
```

Next, register it as a service in your Startup class:

```
services.AddSingleton<ITagHelperComponent, SampleTagHelperComponent>();
```

Now it will automatically be called for both the head and body tags, thus modifying them to include the additional JS and CSS references:

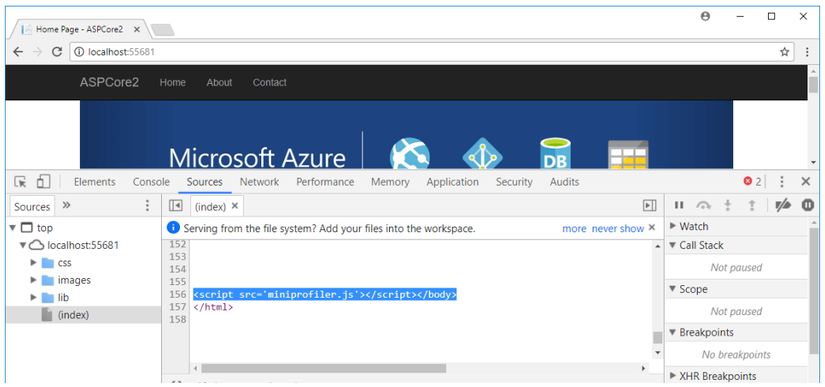


Figure 12, the script added by the TagHelperComponent at the end of body

What happens if you want to modify a tag other than head or body?

Then you need to create a special type of TagHelper that basically targets a given tag name and calls all the registered TagHelperComponents. And you do so by inheriting from the predefined `TagHelperComponentTagHelper` class.

For example, if you wanted to modify the footer component, then you would add the following tag helper:

```
[HtmlTargetElement(“footer”)]
public class FooterTagHelper: TagHelperComponentTagHelper
{
  public FooterTagHelper(
    ITagHelperComponentManager manager,
    ILoggerFactory logger): base(manager, logger) { }
}
```

Now any TagHelperComponent would also be called for footer elements, allowing you to modify them:

```
public override void Process(TagHelperContext context, TagHelperOutput output)
{
  if (string.Equals(context.TagName, “footer”, StringComparison.OrdinalIgnoreCase))
  {
    output.PostContent.AppendHtml(“<p>Modified in a TagHelperComponent</p>”);
  }
}
```

The caveat is that they will be called for **every** footer without distinction, so take this into account when targeting elements that are not unique like head or body! (and double check you really need a TagHelperComponent in such a scenario, and not a regular tag helper, view component, partial, etc)

Application Insights

Finally, you might notice from the earlier discussion about the `Program` class that there is no longer a call to .`AddApplicationInsights`() added to your `WebHostBuilder` (nor is it added for you as part of `CreateDefaultBuilder`)

However, the feature works automatically when either debugging in Visual Studio or running in Azure, how is that possible?

Remember the earlier section about `IHostingStartup`? Application Insights now:

- Is included as part of the .All metapackage
- Provides an IHostingStartup that registered the necessary services
- When running from Visual Studio or Azure the ASPNETCORE_HOSTINGSTARTUPASSEMBLIES will contain the name of the assembly `Microsoft.AspNetCore.ApplicationInsights.HostingStartup`
- A TagHelperComponent is registered to inject the JavaScript needed on the client side

That’s how application insights can work even when it is not referenced anywhere by your app!

That’s interesting, but it might also turn annoying when debugging in visual studio, especially since there is no way of turning the feature off. The only suggested workaround is modifying your launchSettings.json to include the environment variable ASPNETCORE_preventHostingStartup set as True:

```
"environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development",
    "ASPNETCORE_PREVENTHOSTINGSTARTUP": "True"
},
```

Be aware adding this environment variable completely disables IHostingStartup! (But since launchSettings.json is only used in development from Visual Studio, it might be a good compromise unless you have created your own IHostingStartup implementation)

Conclusion

There is no shortage of new features added to the framework on its 2.0 release!

Some like the Razor Pages are a significant new piece added to the framework. I am quite interested in seeing what its acceptance and usage by the wider development community is. Although I haven't used them enough yet to have a better opinion, I can see a combination of pages handling GET request and a typical API controller handling additional client requests working well.

Others like TagHelperComponents, IHostingStartup or IHostedService enable some advanced and interesting scenarios that I am sure at least libraries and tools authors will welcome.

Of course, there is the usual number of smaller improvements and new features that contribute towards a more mature and pleasant framework. Not to forget the release of .NET Standard 2.0 which increases the number of APIs available in .NET Core and might make it viable for teams that were blocked because of missing dependencies.

I am curious to see what will come next! (Although you can always keep an eye on the roadmap)

Download the entire source code from GitHub at
bit.ly/dncm33-aspnetcore-2

Daniel Jimenez Garcia
Author

Daniel Jimenez Garcia is a passionate software developer with 10+ years of experience. He started as a Microsoft developer and learned to love C# in general and ASP MVC in particular. In the latter half of his career he worked on a broader set of technologies and platforms while these days is particularly interested in .Net Core and Node.js. He is always looking for better practices and can be seen answering questions on Stack Overflow.

Thanks to David Pine for reviewing this article.

Ravi Kiran

ANGULAR APPLICATION ARCHITECTURE OVERVIEW

THIS ARTICLE EXPLAINS THE HIGH LEVEL ARCHITECTURE OF AN ANGULAR APPLICATION. THEN IT DISCUSSES THE CORE CONCEPTS OF THE FRAMEWORK WITH SOME SAMPLE CODE SNIPPETS WRITTEN IN TYPESCRIPT.

Angular is a front-end framework created by Google to make the process of building modern web apps, easier.

The capabilities of the front-end web has improved over the last few years, and current gen users demand to see their business data with a rich UI. Angular comes with a number of features to address these needs.

The foundation of Angular is built around a set of core concepts that give strength to the features of the framework. In addition to bringing UI richness, Angular provides ways to structure the code of an application. The framework is designed to build applications using a number of code blocks separated into individual modules.

This approach helps the developers write clean and maintainable code.
In this article, we will understand the high level architecture of an Angular application and go through the core concepts on which the framework is built.

Angular Application - High Level Architecture

An Angular application can be viewed as a tree of components.

The application bootstraps using a component and rest of the application is then rendered with help of a number of sub-components. The following diagram shows how a typical Angular application is divided into components:

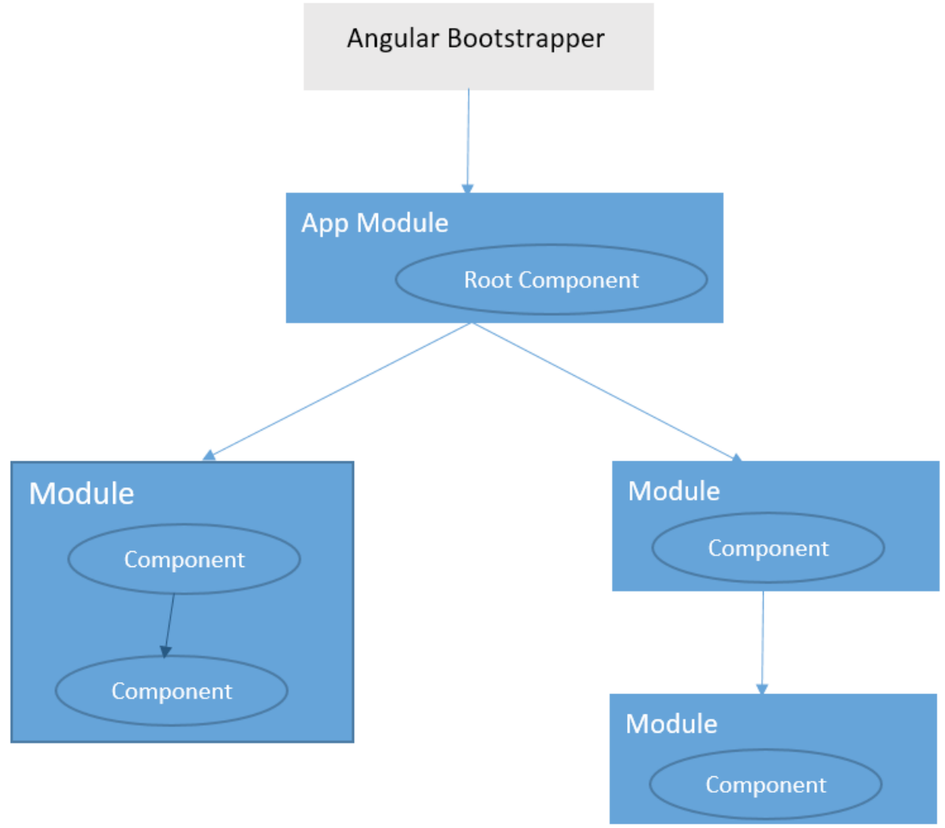


Figure 1 - Angular Architecture

A ***Component*** in Angular comprises of functionality, HTML template and the CSS styles that are used in its template.

As shown in the diagram, every component lives in a module and it is loaded from the module.

The component makes use of ***Services*** to fetch data and present it in a view or to run a piece of logic that can be reused across the application and doesn't involve DOM.

Components receive the objects of services through ***Dependency Injection***.

Directives are used in the template of the component in order to extend the behavior of HTML elements.

The data in the component is presented on the view using ***Data Binding***. Any change in the data is handled by the Change Detection system built in Angular and the UI gets updated to present the latest data.

Zones keep track of the events happening at the browser level to kick in the change detection when anything happens outside Angular's context.

This is how the Angular framework uses its core concepts to handle the applications built on it.

The following figure shows how the different pieces of Angular work together:

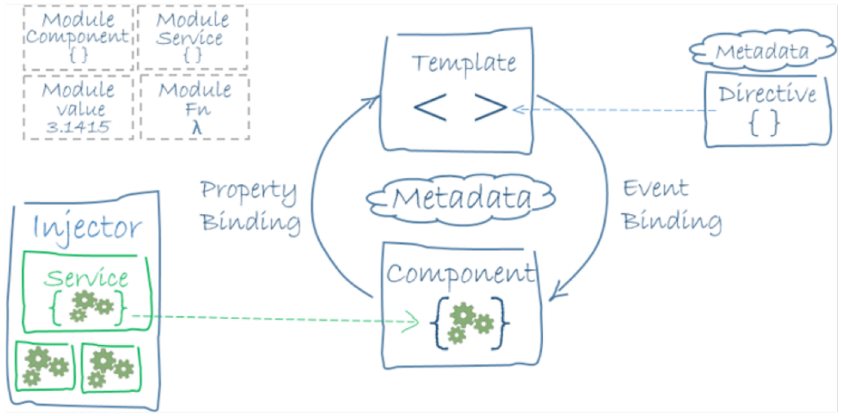


Figure 2 – Overview of an Angular application (source: https://angular.io/guide/architecture)

The next section will detail these core concepts.

Angular Core Concepts

Directives

Directives are used to extend the HTML by creating custom HTML elements and extending the existing elements.

Angular supports three types of directives.

The directives used to create custom elements are called Components. The directives used to extend an HTML element through a new attribute are called Attribute Directives. And the directives that interact with the DOM and manipulate the target element are called Structural Directives.

The components are created using the @Component decorator. The attribute and structural directives are created using the @Directive decorator. The components accept a template, to present it in the custom element. The directives don't have a template, as they are going to act as an add-on to an element.

Components

Generally speaking, a component is a piece of a body, particularly within a machine or a software system.

An Angular application is comprised of different components with different responsibilities. As shown in Figure-1, an Angular application itself starts with a component and this component is used to load the other components.

A component can be anything starting from an element that starts the application, an element that loads content of a page, an element that loads a portion of a page, for example to render a chart to display the details of an item, or anything can be separated into an independent piece of the page.

Angular components are custom HTML elements, which make the application more declarative and the template of the application, more readable. This is because, most of the components have human readable names. If someone uses a bad name for a component, it won't be clear to the reader.

For example, consider the following snippet:

```
<div active="active">
  <ul class="nav nav-tabs nav-justified">
    <li class="tab nav-item active" index="0" heading="Tab 1">
      <a href="" class="nav-link">Tab 1</a>
    </li>
    <li class="tab nav-item" index="1" heading="Tab 2">
      <a href="" class="nav-link">Tab 2</a>
    </li>
    <li class="tab nav-item" index="2" heading="Tab 3">
      <a href="" class="nav-link">Tab 3</a>
    </li>
  </ul>

  <div class="tab-pane">
    <span>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Deleniti,
voluptate, sint odio accusamus excepturi ea doloribus possimus repellendus ut
incidunt ad laborum facere est quae ex itaque expedita. Commodi, beatae.</span>
  </div>

  <div class="tab-pane">
    <span>Aliquid, sequi, deserunt, laboriosam sunt dolore corrupti dolorem
possimus rerum ducimus vel minima porro quo nam. Aliquid, at, impedit, vitae, neque
voluptate doloribus animi reprehenderit nostrum harum libero obcaecati facilis?</
span>
  </div>

  <div class="tab-pane">
    <span>Mollitia, dolores sapiente laboriosam itaque facilis voluptate voluptas
tenetur beatae consequuntur quos corporis harum quas tempore iusto eligendi!
Nihil, est incidunt laborum illo repellat facilis doloribus. Expedita, libero quod
laudantium?</span>
  </div>
</div>
```

This HTML produces a group of three tabs when rendered on a page. Understanding that by simply reading the HTML is difficult, as it consists of a lot of elements and a number of custom CSS style classes. Now imagine the same set of tabs represented as in the following snippet:

Without a doubt, one can say that this version is far more readable and easy to use!

The elements, *tabs* and *tab* are components, which abstract the custom CSS classes and the set of HTML elements to be rendered under the tabs. The JavaScript code required to handle interactions of the tabs is also defined in the component. So, a component can be viewed as a combination of HTML, CSS and JavaScript which work together to perform a task.

In addition to making the application more readable, the components also help in scoping CSS styles. The CSS styles added in a component stay isolated within the component and they don't affect the rest of the page. This feature makes the view of any component more predictable and saves a lot of time for the developers, as they don't need to take extra care to make names of the CSS classes unique.

Angular uses Shadow DOM or an emulator depending on the metadata of the component to keep these styles isolated in the components.

A component is defined in Angular using a TypeScript class with the Component *decorator*.

The decorator accepts metadata of the component, which defines the way a component has to be used in HTML, the component's template and a few other properties. An object of the component class serves as the view model for the template of the component. All of the fields and methods in the component class are accessible in the template.

The following snippet shows the structure of a component:

```
@Component({
  selector: 'sample',
  template: '<div>Some content in the component</div>',
  styles: [`
    div {
      font-style: italic;
    }
  `]
})
class SampleComponent{
  //class definition
}
```

Attribute Directives

An attribute directive looks like any other HTML attribute.

An attribute directive changes the behavior or appearance of the element according to business needs, as it has access to the application's objects. It can interact with the element using events. An attribute directive is a TypeScript class with the Directive decorator.

```
@Directive({
  selector: "[myDummy]"
})
class MyDummyDirective{
  //class definition
}
```

This directive can be used as:

```
<div [myDummy]="prop"></div>
```

The above snippet can be used in the template of a component. This snippet assumes that the component containing this template defines a field named prop. Angular comes with a set of predefined attribute directives to address most of the general purpose needs. They will be discussed in a future article.

Structural Directives

Structural directives modify the template of the element; they can add, remove or replace elements inside the template. Like attribute directives, the structural directives are classes with the Directive decorator.

Usage of the structural directives differs from usage of attribute directives, they are prefixed with an asterisk in the HTML template.

```
<div *repeater="collection"></div>
```

Angular defines a couple of structural directives for common usage. ngIf is the structural directive that can be used to toggle a portion of an HTML template based on a value. The following snippet shows an example:

```
<table *ngIf="data.length > 0">
  <!-- HTML markup of table -->
</table>
```

The above snippet has an HTML table displaying a list of items. The table is added to the page if the variable data has records, otherwise it won't be added to the DOM.

Data Binding

Most data driven applications have a visual interface that allows users to interact with the application by viewing, adding, editing, or deleting data (traditionally referred to as CRUD operations). In general, this visual interface is referred to as the *view*.

For example, to work with your e-mail account, you use an e-mail client that –

-shows you the list of mails in your inbox,
-provides controls to mark them as read, unread, important or delete the mail,
-opens up a mail composer to write a new mail
-and a lot of other controls to perform other operations.

Here the mail client is the view, since the user will use it to view the data in his or her e-mail account.

Now imagine developing a system similar to the e-mail client. It performs a lot of operations with the data and yet displays the most recent data to the user. This cannot be done efficiently unless the framework used to develop the application, has a strong data binding system.

Angular comes with a built-in data binding system.

In an Angular application, data in a component can be used to bind in its view. As mentioned earlier, the methods and properties of a component are directly accessible in the component's view, and the data binding system provides a way to access them.

Angular supports following types of bindings:

- **Property binding**: A property on an HTML element can be bound to the value of a field in the component's class. Whenever value of the property in the component changes, it automatically updates value of the HTML property it is bound to. The following snippet shows a simple example of property binding:

```
<a [href]=”url”>Click here to visit the link</a>
```

Notice the square brackets around the HTML property in the above snippet. They indicate that this field is bound to data.

- **Event Binding**: The events of any HTML element can be bound to public methods of its component class. The method is called whenever the event is triggered by a user's action on the page. The following snippet shows an example of event binding:

```
<button (click)=”close()”>Close</button>
```

- **Two-way Binding**: It is a derived binding that uses both property binding and the two-way binding under the hood. This binding is used to display a value and also to update the value when a new value is entered in the UI. ngModel is a built-in directive that supports two-way binding. The following snippet shows how it is used:

```
<input type=”text” [(ngModel)]=”name” />
```

As you see, the two-way bound directive is enclosed inside [()] (called banana-in-a-box). This notation is again a mix of both property and event bindings.

The parentheses around the event name indicate that it is data bound.

Change Detection

A challenge for anyone building a rich data driven application is to keep the view and its data in sync.

For example, consider a web based e-mail client again.

The view of the inbox has to be updated as soon as the client application receives a notification of a new e-mail. If the application has to perform this task by itself, it needs to listen to the new e-mail notification, build the HTML for the portion of the page displaying the list of e-mails and replace the old HTML with the new one.

A similar set of operations has to be performed to update the count of unread e-mails against the folder names.

Performing these operations manually adds a lot of clutter to the code. It is hard to maintain such code for a longer duration.

The best possible solution to such cases is an **automatic change detection system**!

As the name suggests, the change system detects any change in the state of the objects. Angular comes with a flexible and efficient change detection system. Once it finds any change in the values of the objects used in the view, it asks the framework to run the data binding process and as a result of this, the view gets updated.

The change detection system in Angular knows when to run and the set of actions it has to perform. The change detection system understands three types of objects:

Plain JavaScript objects: The plain old JavaScript objects can be used to build view models in an Angular application. The framework checks for any change in value of the object bound to UI on every browser event and it updates the UI if it finds a change. This technique is called *dirty checking*.

Observable objects: Observables are the JavaScript objects with built-in mechanism to notify a change.

When value of an observable object changes, it publishes an event. One can subscribe to this event and perform an action using the new value set to the object. Angular understands this behavior and it updates the part of page using an observable only when a notification is received. It doesn't manually check for changes to such an object on every browser event.

Angular doesn't provide a way to create observable objects. It internally uses the library RxJS to manage observables in the framework. But it is not limited to RxJS, it can understand any observable object.

Immutable objects: An immutable object is one that gets re-instantiated when any kind of change is made on the object. Angular catches the change made to such objects and updates the portion of the page when a change is available on the immutable object. The portion of the page using an immutable object is not considered for dirty checking. Like observables, Angular doesn't provide a way to create immutable objects. The most popular library to work with immutability in JavaScript at the time of this writing is immutable.js from Facebook.

Any of these objects described here can be used, depending on the need. The interesting part is, different types of objects can be used for different scenarios. An application need not be built using just one type of the object. Usage of observable objects and immutable objects in critical parts of the application boosts performance of the application, as it would have fewer conditions to check.

Angular has to be made aware of the special objects when they are used for data binding. It is done using the *changeDetection* metadata property of the components. The following snippet shows the usage:

```
@Component({
  selector: ‘sample’,
  template: ‘<div>Some content in the component</div>’,
  changeDetection: ChangeDetectonStrategy.OnPush
})
class SampleComponent{
  //class definition
}
```

The component in the above snippet is applied with the OnPush strategy. This strategy tells Angular to not run change detection on this component unless it receives a notification of change on one of its input properties, which can be an observable or an immutable object.

Zones

Any rich JavaScript application has to deal with events like button click, end of timeout, end of an asynchronous XHR request or a WebSocket push. All of these operations are handled asynchronously by the browser using the event loop. The application's data often changes when these operations complete and so the underlying framework has to know when the operation completes.

Zones provide an execution context.

Zones keep track of all the operations performed in the context. Zones can intercept the asynchronous operations. Hence, one can perform some operations before the event starts and after the event ends.

An Angular application runs inside a Zone. This zone is responsible for tracking all of the events happening in the application and to start the change detection process when an event completes.

Because of this, any changes happening on the objects of the application are immediately reflected on the page and the application need not kick in the change detection manually. The intercepting behavior of the zones also allows to correctly profile an asynchronous task and even a group of asynchronous tasks which start at the same time.

Services and Dependency Injection

Say a company is building a sales application and it needs to display a report in the form of a bar chart.

The chart control has to display the report of the sales of a product over the months of a given year. The chart takes the year as an input and it has to fetch the data and then display it on the chart. Being a UI component, the chart control should accept the data and display it. It shouldn't be made responsible to fetch the data to be displayed.

Say, the logic of the fetching the data is handled by a class named SalesData. The chart component needs an object of the SalesData class to get the data and build the chart. If the chart component create an object of the class SalesData by itself, it adds a few problems in the application. The chart control has to be modified every time the signature of the constructor of the SalesData class is modified. It becomes hard to test the chart control, as it depends on a given implementation of an object, which can't be mocked.

This can be solved by the **Dependency Injection pattern**.

Dependency Injection (DI) is a design pattern that solves the problem of handling dependencies needed in a code block.

This pattern says that the dependencies required by a code block have to be created by an external agent and they have to be injected into the code block. As the logic of creating objects is stored in one place, it reduces possibilities of code repetition.

The code consuming an injected dependency doesn't know how the object is created, the only thing it knows is the structure of the object. This means, the object can be easily swapped with any other object that has a similar structure.

We can create a TypeScript interface for the object and have different implementations of the interface for different environments. Say for example, a component uses an object to log messages to the server. If the code is still under development and the team doesn't want to post junk messages on the server, the object responsible for logging the message can be easily swapped with an object that logs the messages to browser's console with methods of similar signatures.

DI is quite popular among server side typed programming languages like C# and Java. AngularJS 1 brought this feature to JavaScript and Angular implements it in an improved way. Angular uses DI to inject services into components, directives, pipes and even in services.

An Angular service is a plain ES6 class that can be used to perform tasks like interacting with an HTTP API, business calculations or anything that doesn't depend on the DOM. In general, the services don't depend much on the block of code where they would be used, so the logic in the service can be used at any place in the application where there is a need of the logic written in the service.

The following snippet shows a service and how it is injected in a component:

```
@Injectable()
class DataService{
  //service definition
}
```

```
@Component({
  selector: 'sample',
  template: '<div>Some content in the component</div>'
})
class SampleComponent{
  constructor(private dataSvc: DataService){}
  //component definition
}
```

The decorator *Injectable* applied to the service class helps in injecting dependencies to the service.

Modules

A module in an Angular application is used to group a set of related directives, services and the other code blocks in a single unit.

We can say that a module is a collection of a set of blocks, which fulfills a piece of functionality. A module can import other modules and can expose its functionality to other modules.

Every Angular application should have at least one module.

Even to start an Angular application, the component to be used at the root has to be declared as the bootstrapping component in the module. The authors of external libraries can use modules to group all of the objects they want to export to the consumers.

A module is a TypeScript class with the decorator *NgModule* applied on it. The following snippet shows a sample module:

```
@NgModule({
  imports: [BrowserModule],  //Modules imported
  declarations: [AppComponent, ApplyStyleDirective, DateFormatPipe],
  //Components, directives and pipes
  providers: [DataService],  //services
  bootstrap: [AppComponent],  //Components to be used to bootstrap application
  exports: [ApplyStyleDirective]  //Components and directives to be exported to
other modules
})
class AppModule {}
```

The above snippet uses a pipe in the declarations property of the module metadata. Pipes are used to format the data before displaying them on the screen.

In addition to providing modularity, modules can also be lazily loaded when they are required in the application. This means the code of the module need not be loaded upfront, instead it can be loaded when the module is required in the application.

***Note**: Angular's modules are different from the modules introduced in ES6. ES6 modules help in keeping the code files small and focused. On the other hand, Angular modules group a set of components, directives, pipes and services in a single unit.*

Conclusion

Angular brings a number of promising features for building rich data driven applications. As discussed in this article, the framework is thoughtfully built using a set of core concepts. The architecture of the framework and the applications built using it depend on these core concepts. Each of the core concept will be explained in detail with examples in future articles. So stay tuned!

Microsoft® MVP Most Valuable Professional

Ravi Kiran
Author

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

Thanks to Keerti Kotaru and Suprotim Agarwal for reviewing this article.

Mahesh Sabnis

What's new in Visual Studio 2017

Initially named "Boston", Visual Studio has gone through a number of transformations in the last two decades. Right from its Visual Basic and Interdev days, Visual Studio has evolved with every release to address new markets and provide improved experiences across multiple platforms and devices.

Visual Studio 2017 (VS 2017) is the 11th version of Visual Studio with a focus on improving experiences around mobile apps, cloud services and devops.

New IDE features like interactive code suggestions (intellisense), easy code navigation, debugging, fast builds and quick deployment are some of the productivity and performance enhancements in VS 2017.

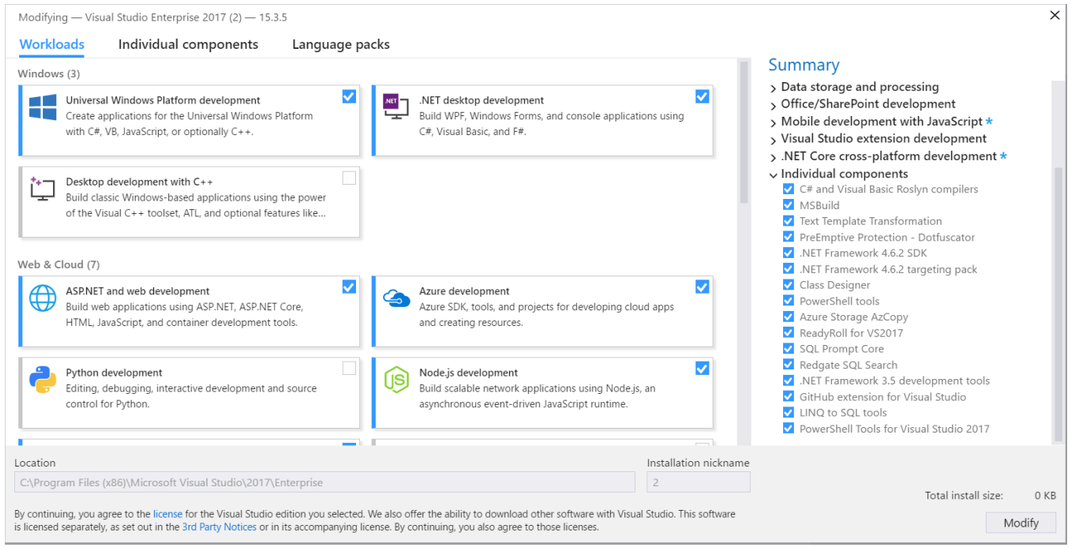
This article explains some of the most important features of VS 2017 which are useful for boosting developer productivity.

Visual Studio 2017 can be downloaded from this link.

Visual Studio 2017 New Features

A new installation and setup experience

Assuming you have downloaded VS 2017, run the setup for a new installation experience, as shown in the following image:



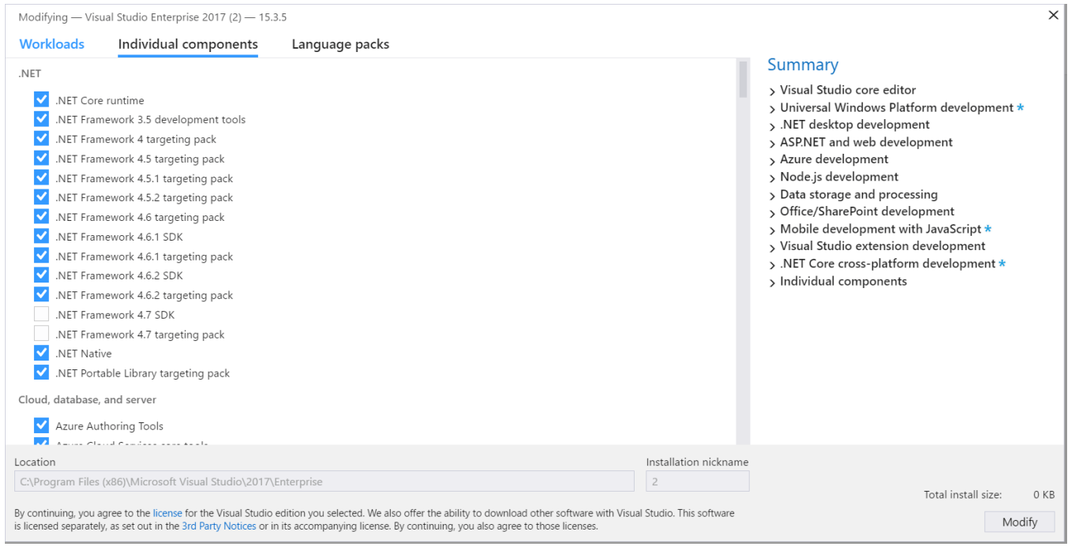
The installation allows you to choose and install just the features which you need; which makes the installation faster.

The **Workloads** tab provides installation options which are grouped to represent common frameworks, languages and platforms. The main advantage of grouping is that a developer can easily choose a feature set to be installed based on their needs e.g. only Windows Features or Web and Cloud features etc.

These group are as follows:

- Windows
- Web and Cloud
- Mobile & Gaming
- Other Toolsets

If you don't want to go the Workloads route, no worries! The **Individual components** tab allows you to pick the components you require, as shown in the following image:



Some options in Individual components are listed here:

- .NET Framework
- Cloud, database, and server
- Code tools
- Compilers, build tools, and runtimes
- Debugging and testing
- Development Activities
- Emulators
- Games and Graphics
- SKDs, libraries and frameworks

The advantage of the new setup experience is that, now one can choose only those features which are needed for his/her development. Individual components can be used by advanced developers to further customize these features of Visual Studio.

The **Language packs** allows you to choose the language for Visual Studio.

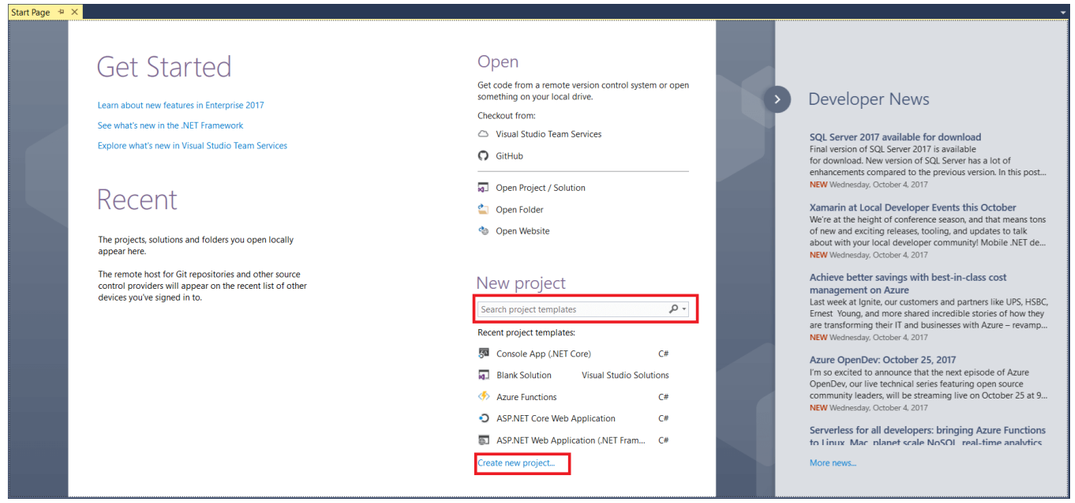
The VS installer, by default, tries to match the language of the OS when it runs for the first time. The following command can be used to force an installer to run in the selected language:

```
vs_installer.exe -locale <Language Token>
```

The following language token are supported by the installer:

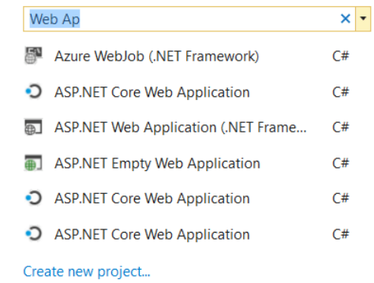
en-Us, zh-cn, zh-tw, cs-cz, fr-fr, de-de, it-it, ja-jp, ko-kr, pl-pl, pt-br, ru-, u, tr-tr,

Once VS 2017 installs and is launched, the start page is displayed as shown in the following image:



The start page shows options like Recent and Open, which are similar to the previous version of Visual Studio e.g. VS 2015, but a new feature worth observing is to create a *New project* using the Search project templates search box or the *Create new project link*.

A new project can be created by entering the project category in the search box as shown in the following image:

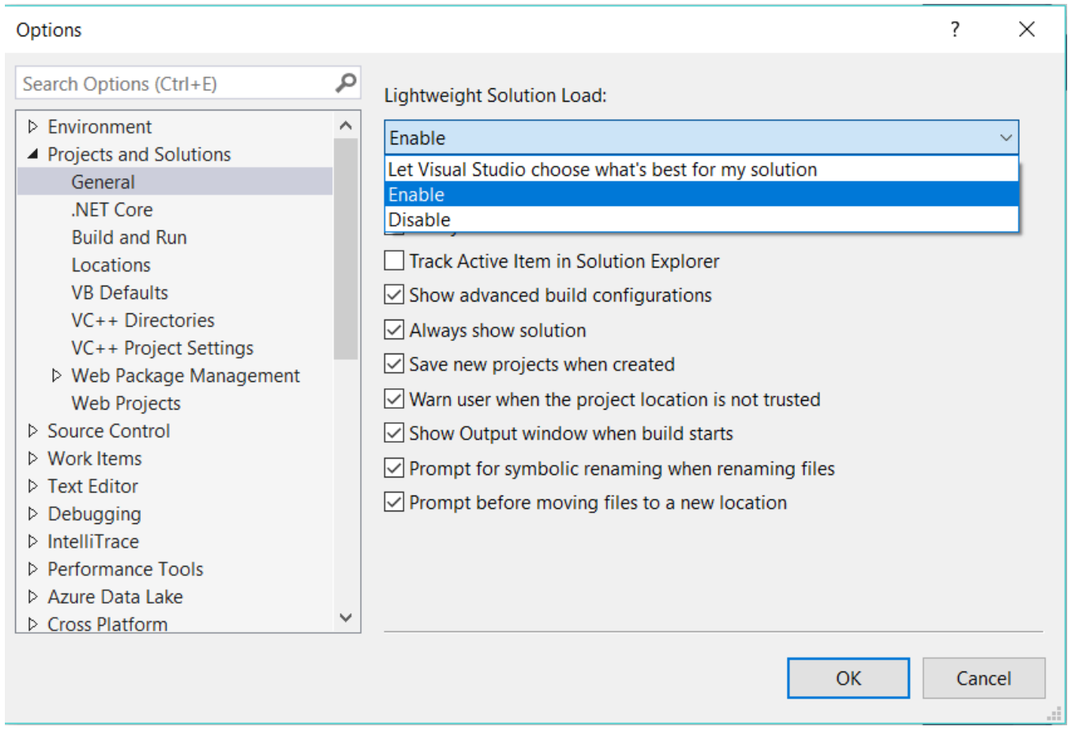


Once the project category (e.g. Web Ap) is entered in the textbox, all matching project templates are searched and displayed. Once the project template is searched and the name of the template is clicked, the New Project window will be opened. The same **New Project** window will also open on clicking the **Create new project** link.

***Note**: If this is not a fresh installation and you have already created projects using VS 2017, then before you start searching for project templates, the recently used project templates are listed.*

Once we create a solution, it will be added in the Recent solution list.

The solution may contain several projects in it, which increases the solution loading time. When the solution is clicked from the Recent list (or else File > Open > Project/Solution), by default, it will be opened with all projects in it. However it is not necessary that the developer will be working on all projects at the same time, so Visual Studio can now be configured to load the solution in a lesser time using Tools > Options > Projects and selecting *Lightweight Solution Load* option as shown in the following image:



The *Enable* option decrease time for loading the solution. This option also results in some limitations e.g. projects will still be loaded as needed when first accessed (directly or indirectly) and in this case, features like edit and continue won't work.

IDE Enhancements for Developers

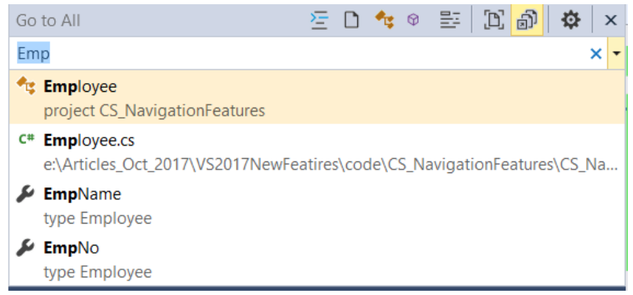
Step 1: Create a Console Application of the name 'CS_NavigationFeatures'. In the project, add a new class file called Employee.cs and add the following code in it:

```
namespace CS_NavigationFeatures
{
  public class Employee
  {
    public int EmpNo { get; set; }
    public string EmpName { get; set; }
    public int Salary { get; set; }
    public string DeptName { get; set; }
    public string Designation { get; set;}
  }
}
```

Open Program.cs and create an instance of the Employee class.

VS 2017 has enhanced navigation experiences for developers to move from one point to another. For e.g. a developer can easily navigate all Employee declarations that matches with the word entered in the search box, which can be opened using a short-cut key Ctrl+T. Using the shortcut brings up a navigation pop-down dialog on the top-right corner of the Visual Studio as shown in the following image:

Enter **Emp** word in the search textbox of the dialog box, which will show all references matching the word *Emp* as shown in the following image:



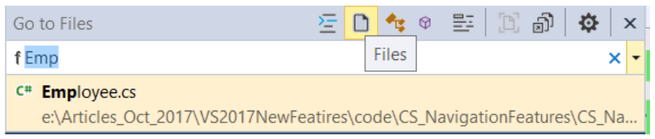
Here a developer can select where to navigate to e.g. to an Employee class, Employee.cs file or other Emp references e.g. members. All this facilitates an easier code navigation, with less distractions.

One of the important features of this dialog is the icon-menu-bar on the top. This icon-menu-bar provides options for selection of code navigation e.g. File, Class, member, etc.

The following image explains it in details.

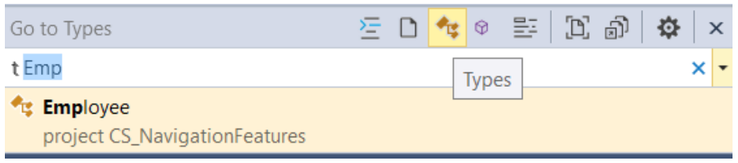
For e.g: to navigate to a file, File Navigation > Symbol is 'f'

..which shows files starting with the matching search criteria:



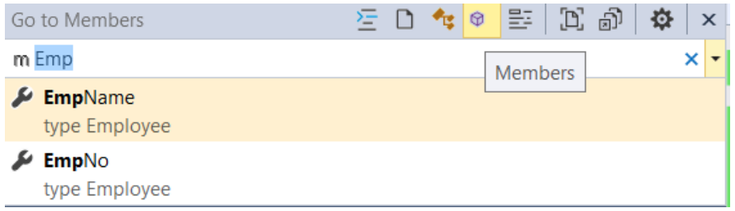
Type Navigation (applied to class, interface) > Symbol is 't'

..shows classes matching with the search criteria:



Member Navigation > Symbol is 'm'

..shows all members matching the search criteria:



Using Reference window

While writing code, developers creates several code files and refer to various types at various places for defining instances e.g. Employee references in various files.

In Visual Studio 2017, these references can be easily searched using an improved reference window.

In the project, add two new class files of name Manager.cs and Payroll.cs and add the following code in it:

Manager.cs

```
namespace CS_NavigationFeatures
{
  public class Manager
  {
    public Employee emp { get; set; }
    public int TravelAllowance { get; set; }
    public int HouseRentAllowance { get; set; }
  }
}
```

Payroll.cs

```
namespace CS_NavigationFeatures
{
  public class Payroll
  {
    Employee emp;
    Manager mgr;
    public Payroll()
    {
      emp = new Employee();
      mgr = new Manager();
    }
  }
}
```

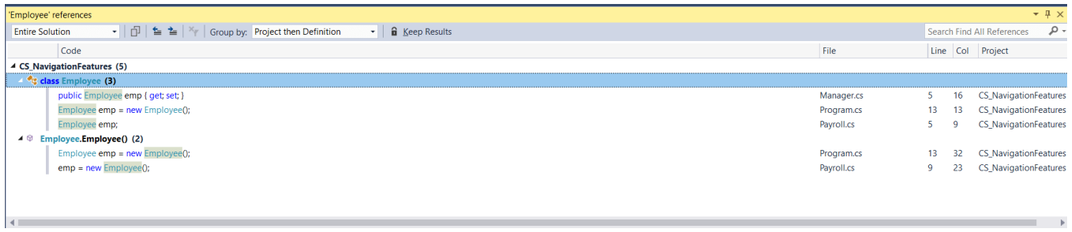
Program.cs is as shown in the following code:

```
namespace CS_NavigationFeatures
{
  class Program
  {
    static void Main(string[] args)
    {
      Employee emp = new Employee();
      Console.ReadLine();
    }
  }
}
```

In the three code files we just saw, an Employee object is used for instance creation and property declaration. Now it is easily possible to find out all the Employee references for the current project. Select Employee declaration in Program.cs as shown in the following image:

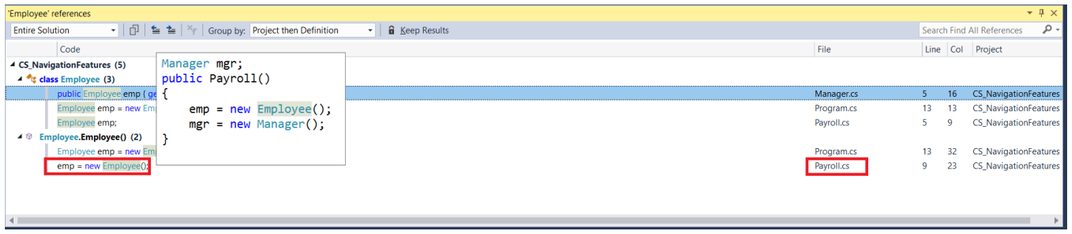
```
static void Main(string[] args)
{
    Employee emp = new Employee();
    Console.ReadLine();
}
```

Now select Shift+F12, and the '**Employee' references** window will be displayed as shown here:



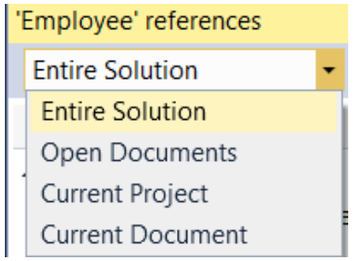
This window will list all Employee references in all the file names.

Just by clicking on the reference, the file can be opened in the VS Editor. The type reference details (in this case *Employee*) can be seen just by moving the mouse cursor on a specific line or reference it as shown in the following image:

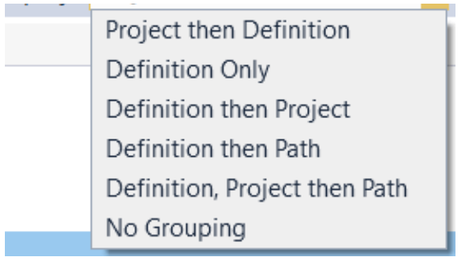


For e.g. in the above image, a reference of an Employee in Payroll.cs is displayed in the tooltip.

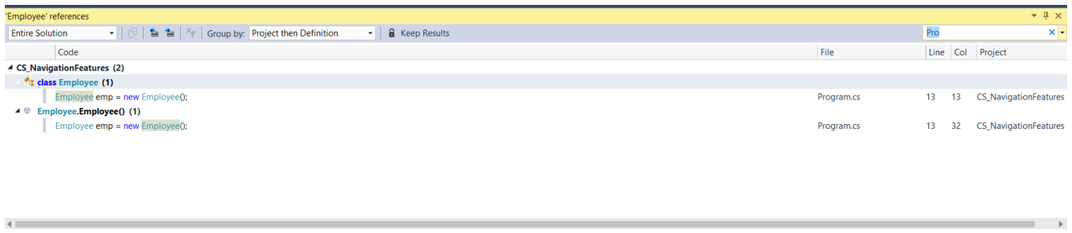
In VS 2015, it was possible to find all references on a class or property and there was no option to filter it. But in VS 2017, these references can be filtered as shown in the following image:



The references can also be grouped as shown in the following image:

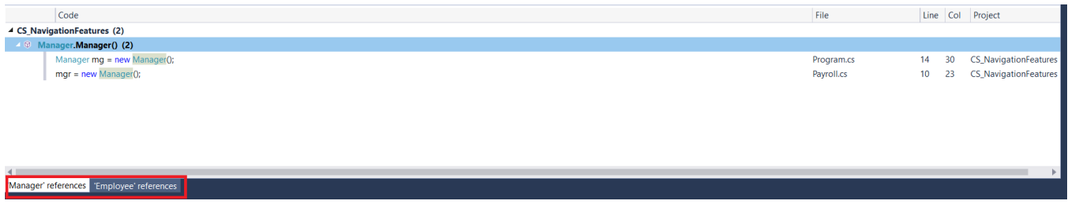


The references window also provides a facility to search from the references using the search textbox on the top-right corner of the window, as shown in the following image. The search can be based on the File name, Column number as well as the Line number.



This shows all Employee references from Program.cs file.

The **Keep Results** toggle button in the references window will make sure that all the references will be saved. The advantage of this feature is that, when new references are searched; instead of overwriting previous references, the new references will be shown in a new tab of the reference window as shown here:



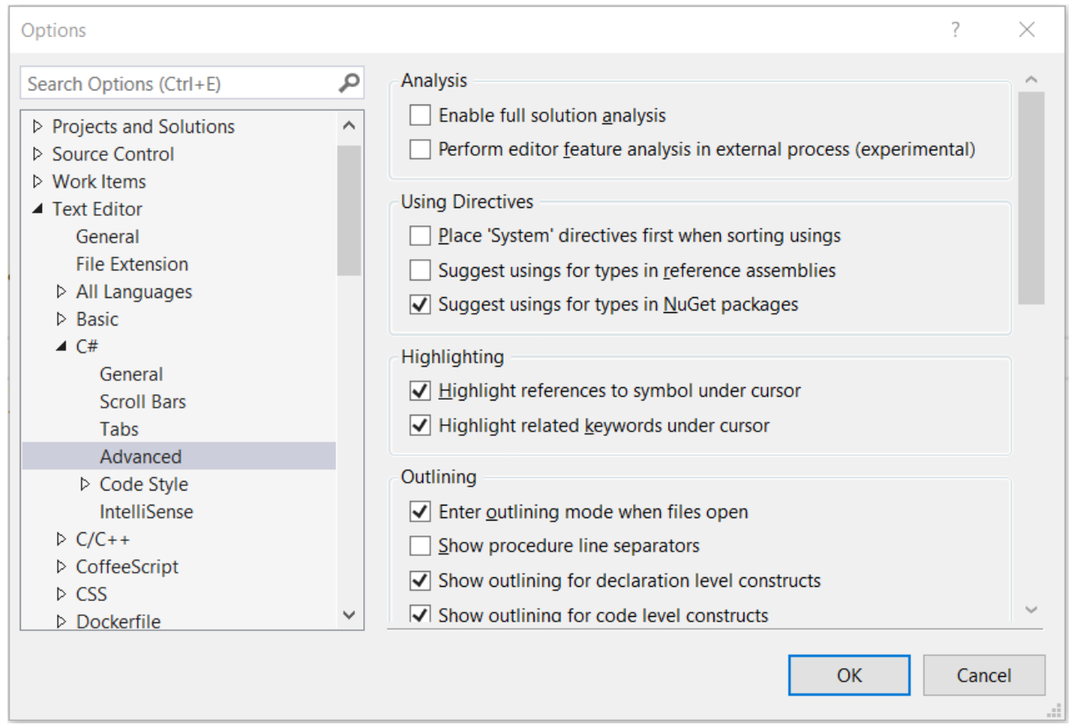
The above image shows references for Manager and Employee.

Automatic Suggestion for NuGet Package for Unknown types

As a developer, when some code is added in the application, sometime it happens that the developer knows the class to be used in the code, but the name of the package from which this class is used, is not known.

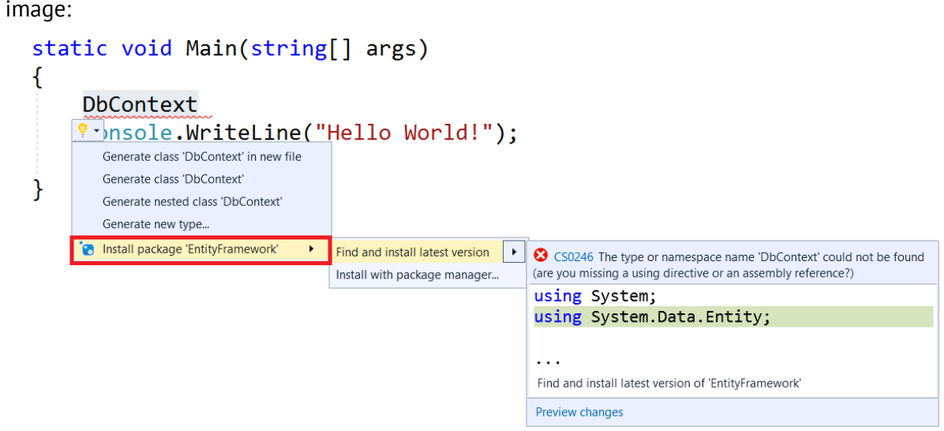
In this case, a developer sometimes searches online for help but such help is not always helpful (pun intended). For e.g. a developer who wants to use the **DbContext** class or a **JObject** class but does not know which NuGet package needs to be installed for the project.

In VS 2017 (also in VS2015.3), there is an option available for S**uggest usings for types in NuGet packages** in the Tools > Options > Text Editor > C# > Advanced as shown in the following image:

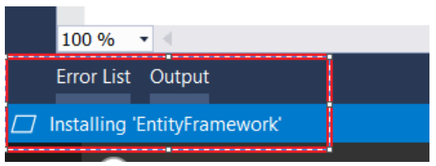


The checkbox for **Suggest usings for types in NuGet packages** is not checked by default, please check it and restart VS 2017.

Open Program.cs and try to use **DbContext** class, select this class name and press **Ctrl+**. Doing so brings up the quick action menu with suggestion for installing **EntityFramework** package as shown in the following image:



Once the **Install package 'EntityFramework'** option is selected, the package will be installed and its status will be shown in the VS 2017.



I hope you too realize what a cool feature this is!

Structure Guide Lines for Current Scope

When the code file contains several classes and a class contains several lines of code with several programming constructs (if...else statements, for loops, etc.), while scrolling down in the VS editor, it is at times difficult to track the scope of the current statement.

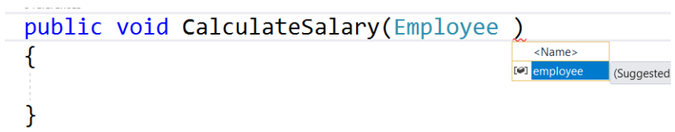
In VS 2017, the code contains dotted lines between curly braces to identify the scope. When the mouse is moved over the dotted lines, the current scope is displayed as shown here:

```
namespace CS_NavigationFeatures
    class Program
        static void Main(string[] args)
            if (true)
    {
        int y = 100;
        y++;
        }
}
```

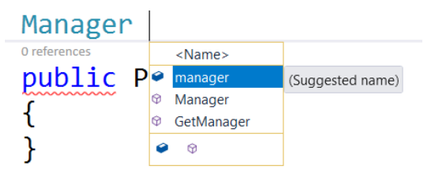
Variable Name/Parameter Name Declaration Suggestion

This is another cool feature of VS 2017. This feature provides suggestion for defining variable name in a class/function as well as input parameters to the function. This helps to follow naming standard for variables/parameters in the application.

To experience this feature, add a **CalculateSalary()** method in the Payroll class in Payroll.cs file. Define the Employee object as an input parameter for this method. VS 2017 will provide name suggestion for the input parameter as shown in the following image:



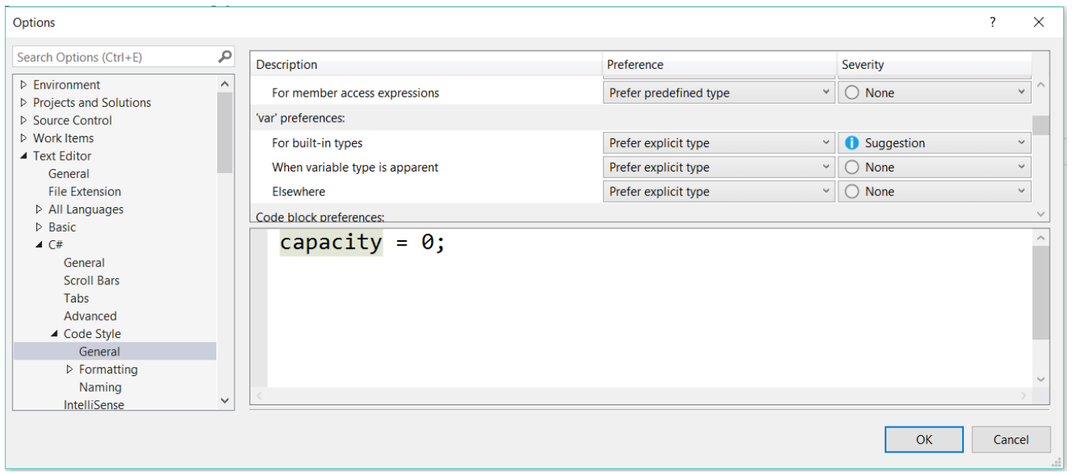
In the Payroll class, try to declare a Manager object. VS2017 will show the suggested name for the declaration of object, as well as for methods too.



This helps to keep coding standards in check.

Coding Style Configuration

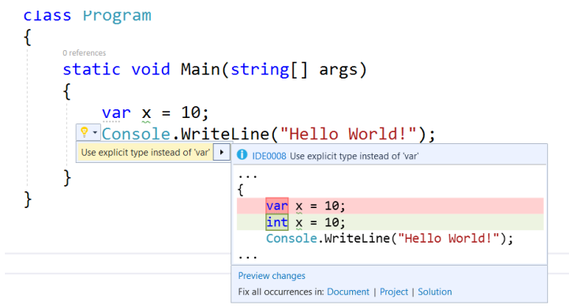
There are some more suggestions settings provided by VS 2017 for validations and coding styles. These suggestions can be set using Tools > Options > Text Editor > C# > Code Style as shown in the following image:



For e.g. in the above image for **'var' preferences**, the **For built-in types** is set to value for **Preference** as **Prefer explicit type** and **Severity** as **Suggestion**. In this case, if the code variable contains declaration as shown here..

```
static void Main(string[] args)
{
    var x = 10;
    Console.WriteLine("Hello World!");
    Console.ReadLine();
}
```

..then the **var** declaration has **dots** below it, which means that there is a suggestion available for the declaration. Press Ctrl+. on **var**, and a quick action menu will display suggestions for the *var* declaration:



Here an explicit type can be selected instead of a *var* declaration.

Likewise, there are several suggestions provided in the options windows which can be helpful to provide a better coding experience. I would encourage you to explore them all.

Improvements in Code-Refactoring

Code refactoring is a major activity that developers must perform while writing code. This makes the code maintainable and readable. In VS 2017, there are some features which provide some great Code-refactoring experiences. Here are some of them:

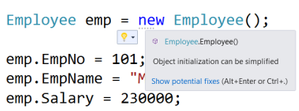
Object Initialization

C# 3.0 already provided a new syntax for object initialization, but consider this code where object properties are initialized in a manner as shown here:

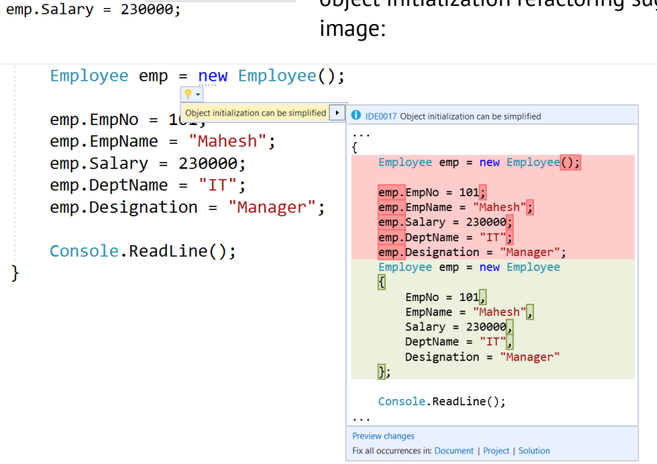
```
static void Main(string[] args)
{
    Employee emp = new Employee();

    emp.EmpNo = 101;
    emp.EmpName = "Mahesh";
    emp.Salary = 230000;
    emp.DeptName = "IT";
    emp.Designation = "Manager";

    Console.ReadLine();
}
```

To change the above code using the Auto-Initialized syntax of C# 3.0, VS 2017 provides a new feature.

Observe the underlined dots below the **new** keyword for object creation. Now move your mouse cursor on the *new* keyword, to bring up a pop-up that shows the message **Object initialization can be simplified:**



Click on the down-arrow of the quick action menu, to bring up the new object initialization refactoring suggestions as shown in the following image:



Select **Object initialization can be simplified** and you will find that the code will be refactored as below:

```
Employee emp = new Employee
{
    EmpNo = 101,
    EmpName = "Mahesh",
    Salary = 230000,
    DeptName = "IT",
    Designation = "Manager"
};
```

How convenient! We could change the object declaration/initialization in a jiffy!

Managing *null* value check for the reference of input parameters of method

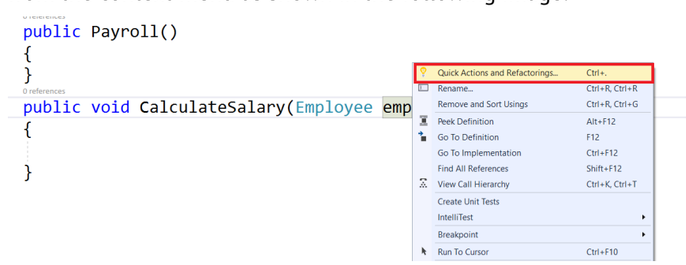
When a method accepts a reference type (e.g. Employee object) as an input parameter, then while accessing that method, it is important to check for a ***non-nullable*** value for the input parameter, before passing it to the method. Alternatively, the method must have code for checking for a ***null*** value.

In VS 2017, this feature is provided so that a method can be added with the required ***null*** check.

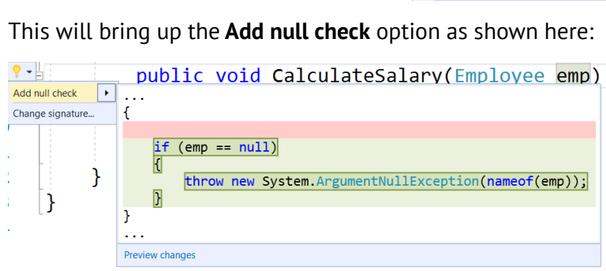
In the Payroll class in Payroll.cs file, add the CalculateSalary() method as shown in the following code:

```
public void CalculateSalary(Employee emp){}
```

To check the ***null*** value for the *emp* object, right-click on emp and select ***Quick action and Refactorings option*** from the context menu as shown in the following image:



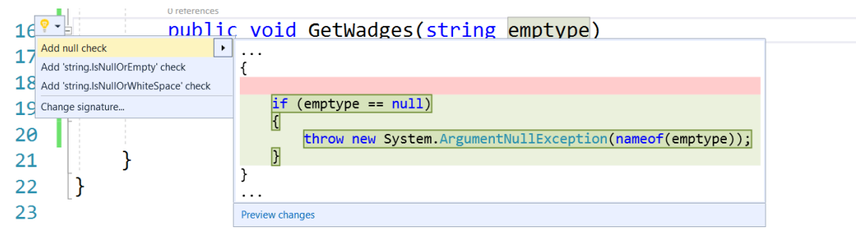
This will bring up the **Add null check** option as shown here:



Once this option is selected, the method will be added with a null check condition:

```
public void CalculateSalary(Employee emp)
{
    if (emp == null)
    {
        throw new System.ArgumentNullException(nameof(emp));
    }
}
```

Likewise, if the method accepts a string parameter, the code can be refactored to generate conditions for **string.IsNullOrEmpty** and **string.IsNullOrWhiteSpace** check in the method.



This useful feature helps developers to reduce logical errors in the code.

Changing the method signature

When a method having various input parameters in a specific order needs to changed either by removing some parameters or changing order of parameters, then it will definitely result in to a compilation error.

If the method is called at various places in code, the Developer would need to manually keep track of changes in the method and update the code accordingly.

This is a time-consuming task!

In VS 2017, the signature of the method can be easily changed and these changes can be reflected at all places where this method is called. This saves the developer some precious time and makes the code error-free.

Consider the following PrintName() method in the program class of the Program.cs file.

```
class Program {
  static void Main(string[] args)
  {
    PrintName(“Mahesh”,”Sabnis”);
    Console.ReadLine();
  }
  static void PrintName(string fname, string lname)
  {
    Console.WriteLine($”First Name {fname} Last Name {lname}”);
  }
```

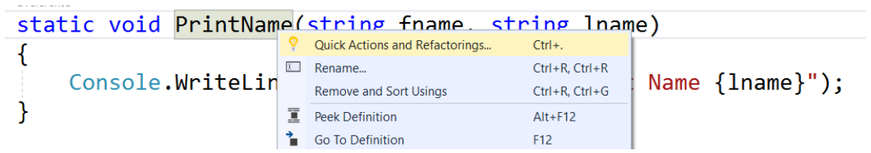
The **PrintName()** method accept two string parameters fname and lname.

This method is called in the **Main()** method with the values set for **fname** and **lname** as Mahesh and Sabnis respectively.

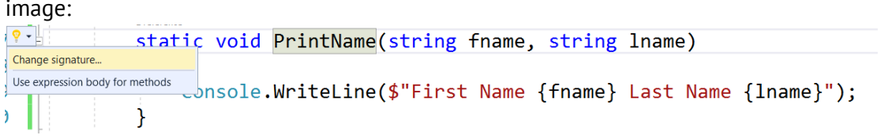
If we need to change an order of fname and lname in the PrintName() method, then in the Main() method, values for these parameters needs to be changed manually.

In VS 2017, this can be done easily by refactoring the method.

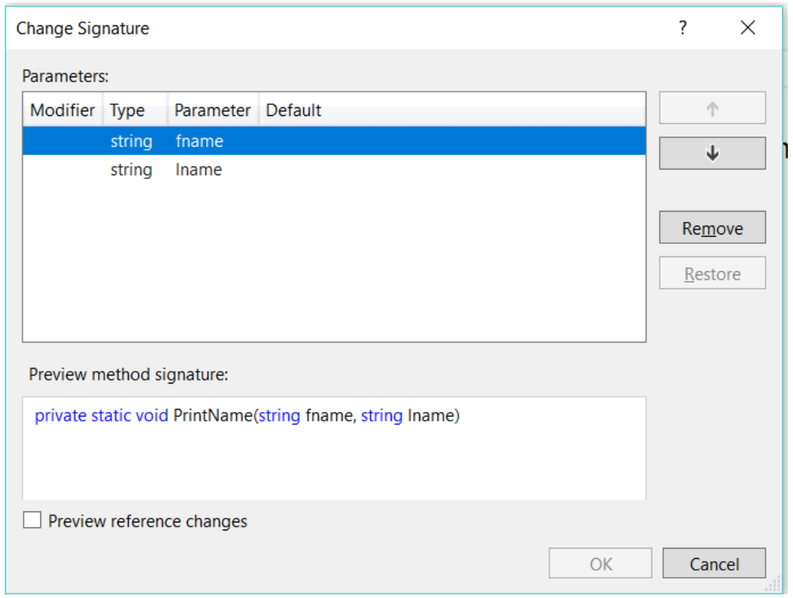
Right-click on the method name and select **Quick action and Refactorings** option from the context menu.



Doing so will show the Quick Action menu with a Change signature option as shown in the following image:



Clicking on the **Change signature** option will show a Change signature window.



Using this window, the order of parameters in a method can be changed or they can be removed altogether.

In the image you just saw, click on the button with the downward arrow and move the **fname** parameter below the lname parameter. Now click on the Ok button, and you will see that the method signature and its call will be updated.

```
static void Main(string[] args)
{
    PrintName("Sabnis", "Mahesh");

    Console.ReadLine();
}
1 reference
static void PrintName(string lname, string fname)
{
    Console.WriteLine($"First Name {fname} Last Name {lname}");
}
```

Once the PrintName() method signature is updated, the PrintName() method call with its parameter values is also updated.

The advantage of this feature is that it reduces potential errors which a developer can make while updating the method definition in the application.

Debugging and Exception Handling Improvements

Once the code has been written, it is important for a developer to debug the code to ensure that the code is logically working and has been diagnosed for exceptions, if any.

In VS 2017, there are some improvements made for debugging and exceptions management.

Debugging Run to Click

In previous versions of Visual Studio, when breakpoints were applied on various lines, it was necessary for developers to press F5 to skip lines in between two breakpoints, to continue debugging from next breakpoint.

Or, for the developer to skip the current breakpoint and continue debugging ahead, it was necessary to put additional breakpoints. In this case, it was necessary for the developer to keep track of all breakpoints (imagine a for/foreach loop occuring in between the code).

To get rid of this, in VS 2017, a new feature known as **Run to Click** is added (similar to **Run to cursor** feature of the previous version), which allows the developer to skip the lines in between breakpoints and start executing code from the desired line.

Consider the following image. A breakpoint is applied on the PrintName() method, so the debugger will stop at this method.

```
static void Main(string[] args)
{
    PrintName("Sabnis", "Mahesh");
    Console.WriteLine("For Reverse");

    Reverse("Mahesh Ramesh Sabnis");
    Numbers n = null;
    Divide(n);

    Console.ReadLine();
}
```

If the developer does not want to stop an execution at the **PrintName()** method and wants to continue execution directly from the **Reverse()** method call, just move the mouse cursor adjacent to the Reverse() method, and a **Run Execution to here** symbol will be displayed as shown in the following image:

```
static void Main(string[] args)
{
    PrintName("Sabnis", "Mahesh");
    Console.WriteLine("For Reverse");

    Reverse("Mahesh Ramesh Sabnis");
    Run execution to here = null;
    Divide(n);

    Console.ReadLine();
}
```

Click on this symbol, and all the lines of code in between PrintName() call to Reverse() method calls will run past it, and the Reverse() method call will be highlighted as shown in the following image:

```
static void Main(string[] args)
{
    PrintName("Sabnis", "Mahesh");
    Console.WriteLine("For Reverse");

    Reverse("Mahesh Ramesh Sabnis");
    Numbers n = null;
    Divide(n);

    Console.ReadLine();
}
```

This means that an execution is continued from the Reverse() method call, hence there is no need to apply a breakpoint at the Reverse() method call.

The New Exception Information Helper

Dealing with exception is a common developer problem irrespective of the technology.

Figuring out the reason for an exception is one of the most frustrating experiences. In previous Visual Studio versions, exception information messages used to be displayed as shown here:

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ExceptionInfo
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Numbers n = null;
            Divide(n);
            Console.ReadLine();
        }
        1 reference
        static void Divide(Numbers m)
        {
            var res = m.X / m.Y;
        }
    }
```

NullReferenceException was unhandled
An unhandled exception of type 'System.NullReferenceException' occurred in Exception.exe
Additional information: Object reference not set to an instance of an object.
Troubleshooting tips:
Check to determine if the object is null before calling the method.
Use the "new" keyword to create an object instance.
Get general help for this exception.
Search for more Help Online...
Exception settings:
Break when this exception type is thrown
Actions:
View Detail...
Copy exception detail to the clipboard
Open exception settings

In this case, it was challenging for the developer to find out the exact exception and it was necessary to see the **View Details** and Inner **Exception** to find out what is the reason behind the exception.

In VS 2017, a new **non-modal** exception handling helper is provided as shown here:

```
1 reference
static void Divide(Numbers m)
{
    var res = m.X / m.Y;
}
```

Exception Unhandled
System.NullReferenceException: 'Object reference not set to an instance of an object.'
m was null.
View Details | Copy Details
Exception Settings
Break when this exception type is thrown
Except when thrown from:
CS_NavigationFeatures.exe
Open Exception Settings | Edit Conditions

This helper shows an exception message and provides the reason behind the exception. In this case the **Object reference is not set to an instance of an object**, exception is fired and the cause of the exception is also displayed as **m as null**.

This helps developers to understand what the reason behind an exception is and how to handle it. The main advantage here is that, earlier it was required for the developer to see the **Inner Exception** in the **View details** dialog; however in the VS 2017, the in-detail exception information is seamlessly provided to the developer.

Live Unit Testing

Apart from all the above features discussed, there is one more important enterprise feature provided in VS 2017, and that is **Live Unit Testing**.

This feature encourages Test Driven Development (TDD) and helps in developing better quality code and code coverage. You can read more about Live Unit Testing at http://www.dotnetcurry.com/visualstudio/1363/live-unit-testing-visual-studio-2017.

Conclusion

Visual Studio 2017 is packed with improvements and new features that increase productivity. This all-new IDE released by Microsoft helps developers to build stunning apps on Windows and other platforms.

I hope this article served its purpose and introduced you to these productivity features which will help you simplify your most common tasks, and ultimately boost your productivity

Microsoft® MVP Most Valuable Professional

Mahesh Sabnis
Author

Mahesh Sabnis is a DotNetCurry author and Microsoft MVP having over 17 years of experience in IT education and development. He is a Microsoft Certified Trainer (MCT) since 2005 and has conducted various Corporate Training programs for .NET Technologies (all versions). Follow him on twitter @maheshdotnet

Thanks to Damir Arh and Suprotim Agarwal for reviewing this article.

JAVASCRIPT

ES8 / ES 2017

What's New?

Keerti Kotaru

TC39 proposals and Stage definitions

ECMA Script/ES is a language specification created to standardize JavaScript.

ES5, the fifth version of the specification and released in 2009, enabled the language with important and powerful features. It allowed sophisticated UI applications built with JavaScript. ES6 onwards, ECMA script releases have been once a year.

ES6 is also called ES2015. ES7 is ES2016 and ES8 is ES2017.

The Ecma TC39 committee is responsible for evolving ECMAScript. TC39 uses the following stage definitions to move a JavaScript feature to one of the ECMA Script releases. Features/specifications in stage 4, in time for each year's release, are included in the release.

Stage 0	Strawman. New specification.
Stage 1	Proposal. Demonstrate need for the addition, solution and challenges.
Stage 2	Draft. Specifics on syntax and semantics
Stage 3	Enable users to experiment and provide feedback. Can be refined further based on feedback.
Stage 4	Ready to be included in ECMA Script

ECMAScript 2018 (ES8) New Features

Following are the new features in ES8/ES2017.

async/await

Context

JavaScript is asynchronous. Traditionally, callbacks were passed and invoked when an asynchronous operation returned with data. But in this model, callbacks invoked callbacks which invoked more callbacks that created a *callback hell* scenario.

Promises largely solved this problem with each asynchronous operation/function returning a promise object. A promise could be resolved successfully with data. And in case of a failure, it got rejected with error information.

Following is a Promise constructor introduced in ES6. Before this API was introduced, ES5 used promises from libraries like JQuery

```
const getData = () => {
  return new Promise( (resolve, reject) => {
    // resolve after 3 second timeout.
    setTimeout(() => resolve({ key: new Date().getTime()}), 3000 );
    });
}
```

The `getData` function creates and returns a promise object. It has two function callbacks passed in as parameters: resolve and reject. The sample we just saw invokes resolve, once it times out.

The following `processData` function invokes `getData`. It returns a promise. When the promise is resolved, `processData` can utilize the JSON retrieved.

In this basic sample, let's console.log it.

```
function processData(){
  getData().then((data) => console.log(data));
};
```

Imagine `processData()` needs to return data with additional modifications done to it. Without async/await, which will be detailed shortly, we will have to create another Promise, modify the data and then resolve it.

Here's how the processData function returns a new promise.

```
function processData() {
  return new Promise((resolve, reject) => getData()
    .then((data) => resolve( {ticks: data.key || 0})));
}
```

An **async** function simplifies this syntax. Use `await` keyword to assign data to a variable when a promise is resolved. A function using `await (parent`) needs to be marked `async`.

Consider the following code snippet for the syntax where the function starts with the *async* keyword.

```
async function processData() {
  // getData() function returns a promise.
  let data = await getData();
  return { “ticks”: data.key || 0 };
}
```

For an arrow function syntax, consider the following code snippet. Notice the use of `async` keyword before the parameter list.

```
const processData = async () => {
  let data = await getData();
  return { “ticks”: data.key || 0 };
}
```

Let’s print the object returned from processData.

Note: *Most samples in this article could be run on a browser prompt, like Google Chrome console. They are simple enough to be legible on the browser console. If you feel comfortable to write it in a separate JavaScript file, you may do so.*

```
console.log(processData());
```

(Output on Google Chrome)

```
• Promise {[[PromiseStatus]]: "pending", [[PromiseValue]]: undefined}
    1. __proto  :Promise
    2. [[PromiseStatus]]:"resolved"
    3. [[PromiseValue]]:Object
        1. ticks:1505624296442
        2. __proto  :Object
```

As you can see, it returned a promise implicitly.

We may use the `then` callback to print the data returned

```
processData().then((data) => console.log(data));
```

Or use `await`.

```
(async () => console.log(await processData()))();
```

Because every function using `await` needs to be marked `async`, we are using a self-executing function here.

Errors in async function

An error in `async` function rejects the returned promise. Consider the following sample.

Notice the errorAsync function throwing an Error object. It results in an error callback (of the promise) invoked.

Browser Support

The following table depicts browser support by version (when the support started). Keep in mind, this may change with potentially more browsers supporting the feature. This information is as of 1st Oct 2017 (at the time of authoring the article).

	Chrome	Edge	Firefox	IE	Opera	Safari
Desktop	55	Yes. Version # unknown	52	Unknown	42	10.1
Mobile	55	Yes. Version # unknown	52	Unknown	42	10.1

Editorial Note: *Please use the “Links and References” section at the end of this article for the most current browser support information.*

Object.entries

The API `Object.keys` has been available since the ES5 days (with certain improvements in ES6). It returns all keys of an object. Complementing this API, `Object.entries` is added to the ECMA Script proposal to enable an object to be used with an iterator.

`Object.entries` returns an array of key/value pairs for a given object. Consider the following sample.

One of the use cases for the API is that the returned entries could be iterated through by creating a map object.

Please note `Object.entries` returns an array for only the enumerable properties of an object.

Browser Support

	Chrome	Edge	Firefox	IE	Opera	Safari
Desktop	54	Yes. Version # unknown	47	No Support	No Support	10.1
Mobile	54	Yes. Version # unknown	47	No Support	No Support	10.1

Object.values

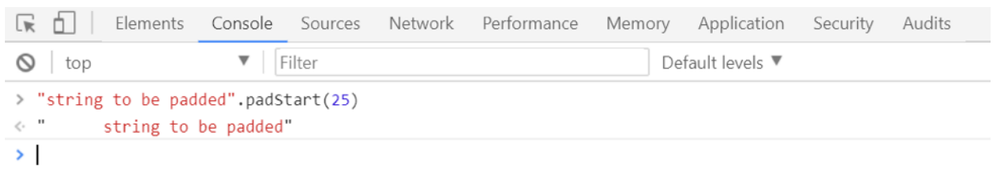
Complimenting Object.keys and Object.entries, the Object.values also returns enumerable values of a JavaScript object. It allows an object to be used with an iterator.

Browser Support

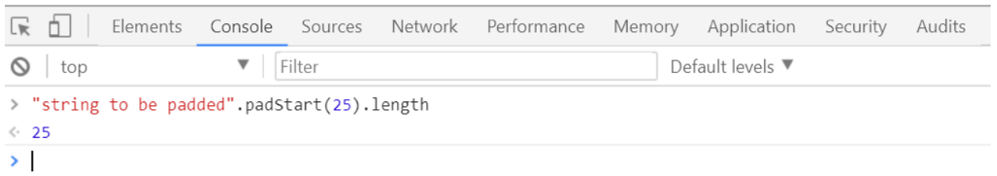
	Chrome	Edge	Firefox	IE	Opera	Safari
Desktop	54	Yes. Version # unknown	47	No Support	Yes, Version # unknown	10.1
Mobile	54	Yes. Version # unknown	47	No Support	Yes, Version # unknown	10.1

String.prototype.padStart

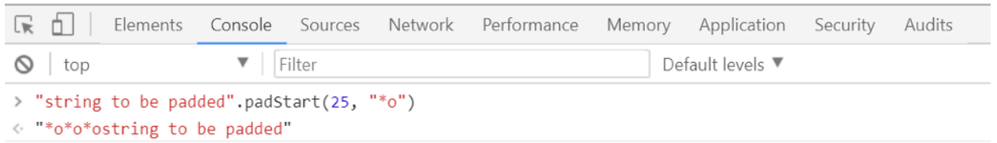
This is a new string handling function in ES2017 to pad the given string at the beginning. Consider the following sample:



The resultant string will be of the length specified.



In the sample we saw, the string is padded with a space (U+0020). This is the default character. We can pass an additional parameter with the string to be padded at the start.

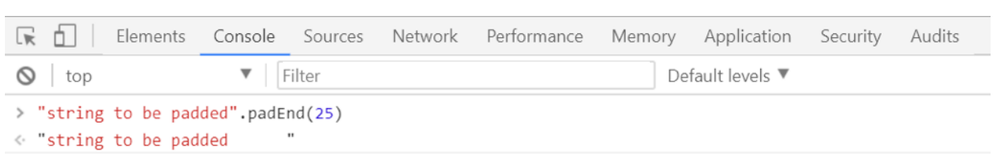


Browser Support

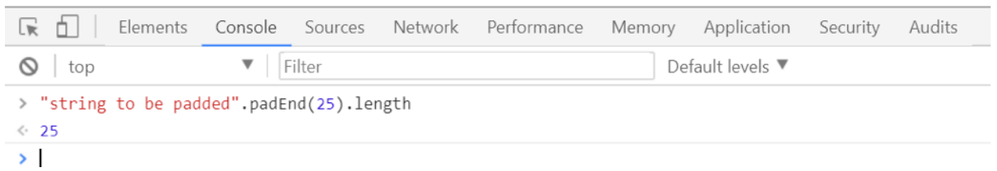
	Chrome	Edge	Firefox	IE	Opera	Safari
Desktop	57	15	48	15	44	10
Mobile	57	Yes	48	No Support	Unknown	10

String.prototype.padEnd

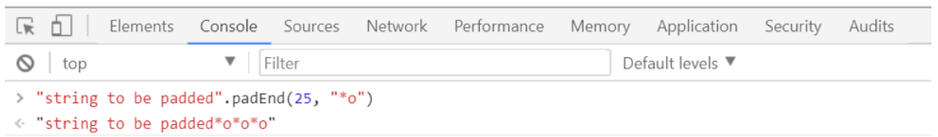
This is another new String handling function to pad a given string at the end. Consider the following sample:



The resultant string will be of the length specified.



In the sample above, the string is padded with a space (U+0020), which is the default character. We can also pass an additional parameter with the string to be padded at the end.



Browser Support

	Chrome	Edge	Firefox	IE	Opera	Safari
Desktop	57	15	48	15	44	10
Mobile	57	Yes. Unsure about version #	48	No Support	Unknown	10

Trailing commas in function parameters

You can now allow a syntax to end the parameter list of a function with a comma (followed by no parameter). Consider the following,

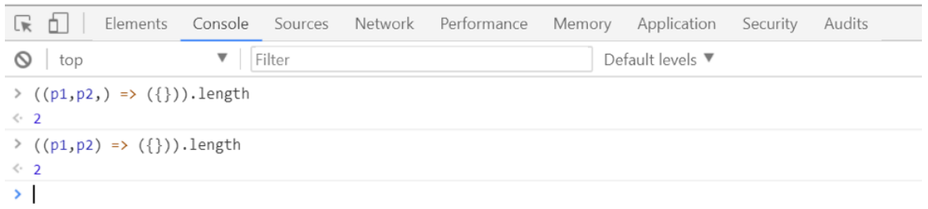
```
const func = (p1,
  p2,
  ) => `parameter1: ${p1}. parameter2: ${p2}`;
```

This is to help semantics with source control products. Consider the following definition.

If a function parameter p3 is added by a team member, adding comma after p2 will annotate this line too that is changed by a team member. Allowing trailing comma will let the new line be added without touching the line with p2.

```
var func2 = (p1,
    p2
    ) => `parameter1: ${p1}. parameter2: ${p2}`
```

This doesn't affect `length` on function object.



getOwnPropertyDescriptors

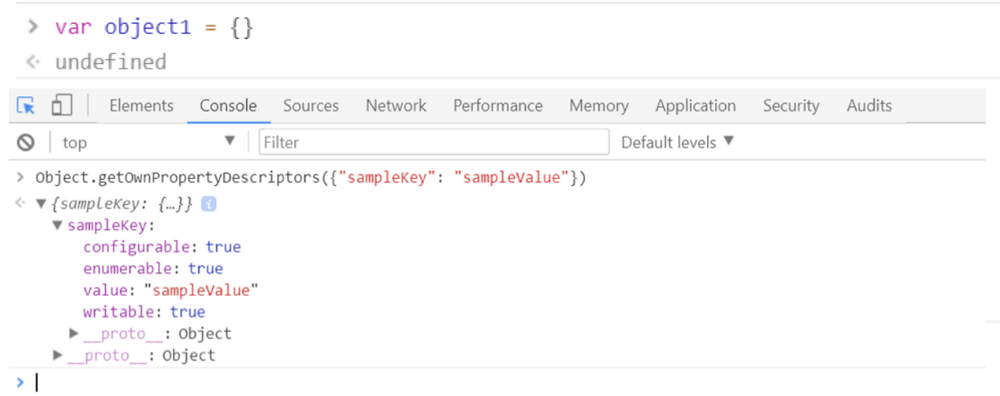
This method returns property descriptors of a given object. Descriptors include the following information about each property on an object,

- Value – value associated with the property.
- Writable – If true, attribute can be read from and written to. If not, will be a read only property.
- Get – getter function for the property
- Set – setter function for the property.
- Configurable – if true, property descriptor may be modified and property is allowed to be deleted.
- Enumerable – If true, property shows up while enumerating the object.

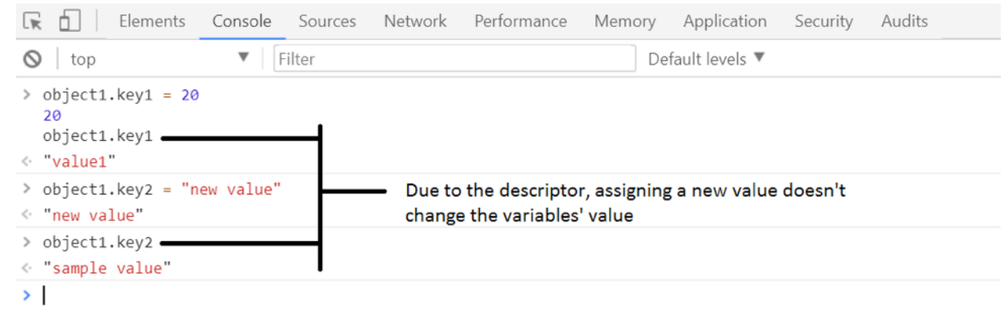
Refer to the following snippet. It shows a descriptor on a `sampleKey` of an object.

We often need to use immutable JavaScript objects. Many frameworks (like Redux for application state) require objects to be immutable. To ensure immutability, we often clone an object. There was no clean way to clone an object in JavaScript.

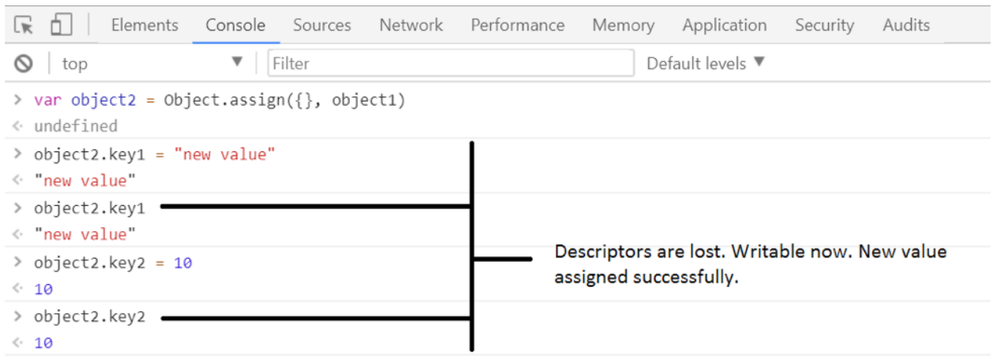
`Object.assign` is a common API used to clone an object. Consider the following code:



Notice key1 and key2 are defined with a descriptor object. It allows us to provide additional descriptor metadata. For example, we can't change the values of key1 and key2 as it's not writable and no setter defined.

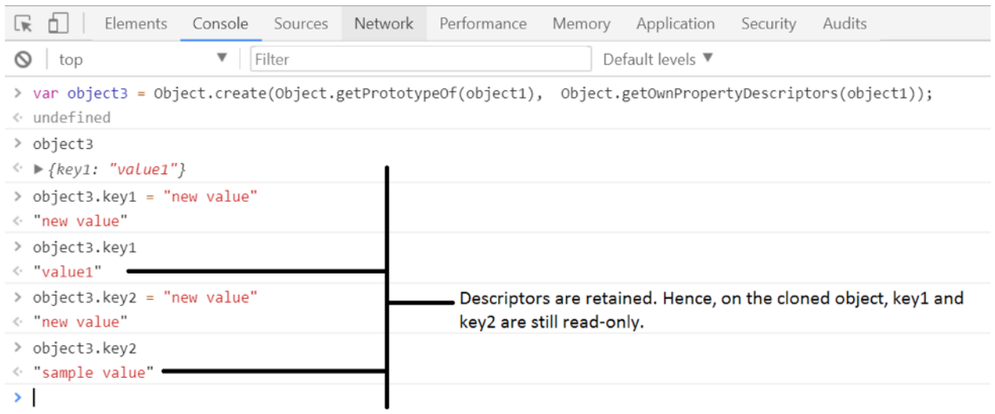


Using Object.assign for the above example will clone object1 but the descriptors' metadata is lost.



With the use of `getOwnPropertyDescriptors`, cloning can be done along with the descriptors.

Consider the following snippet:



Finally, compare descriptors information among the three objects object1, object2 and object3. Notice object1 and object3 have similar descriptors considering cloning included descriptors.

```
> Object.getOwnPropertyDescriptors(object1)
▼ {key2: {…}, key1: {…}}
    ▶ key1: {value: "value1", writable: false, enumerable: true, configurable: false}
    ▶ key2: {set: undefined, enumerable: true, configurable: false, get: ƒ}
    ▶ __proto__: Object
> Object.getOwnPropertyDescriptors(object2)
▼ {key2: {…}, key1: {…}}
    ▶ key1: {value: "new value", writable: true, enumerable: true, configurable: true}
    ▶ key2: {value: 10, writable: true, enumerable: true, configurable: true}
    ▶ __proto__: Object
> Object.getOwnPropertyDescriptors(object3)
▼ {key2: {…}, key1: {…}}
    ▶ key1: {value: "value1", writable: false, enumerable: true, configurable: false}
    ▶ key2: {set: undefined, enumerable: true, configurable: false, get: ƒ}
    ▶ __proto__: Object
```

Browser Suppo

	Chrome	Edge	Firefox	IE	Opera	Safari
Desktop	54	Yes (no information when the support started)	50	No Support	41	10
Mobile	54	Yes (no information when the support started)	50	No Support	Unknown	Unknown

SharedArrayBuffer

It enables creation of a shared location to use between agents. While we create web workers, data can be shared with `postMessage` call. Before `SharedArrayBuffer`, there is no common memory between threads.

`SharedArrayBuffer` enables using a common memory location between threads. Here's a code snippet which creates a worker and posts a message.

```
var work1 = new Worker('workerFile.js');
var msg = {"key1": "sample value"};
work1.postMessage(msg);
```

Here's the worker snippet:

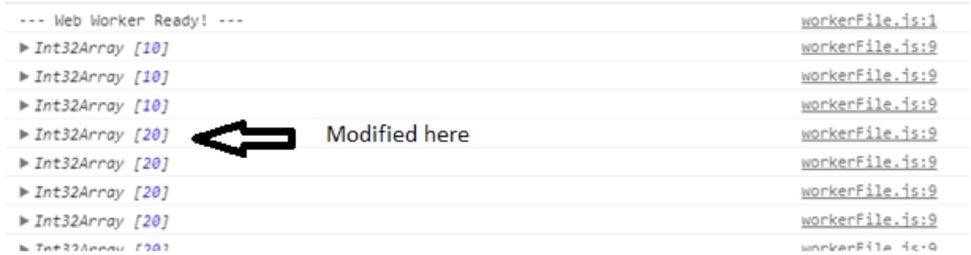
```
onmessage = (event) => {
       console.log(event.data);
};
```

Note that changing the message object doesn't reflect in the worker file. We could use `SharedArrayBuffer` to have a common location that gets updated in the worker when modified in the main thread.

Consider the following piece of code which creates a worker, and creates `SharedArrayBuffer` with Int32Array (view). It will first post the data to the worker (thread). The worker is expected to print data in a loop. After couple of seconds, the main thread will update the data.

```
// Create Worker
var work1 = new Worker('workerFile.js');

// Create SharedArrayBuffer
var sab = new SharedArrayBuffer(4);
var ia = new Int32Array(sab);

// set value
ia[0] = 10;

// post data to the worker
work1.postMessage(ia);

// after 2 seconds update the posted object.
setTimeout( () => ia[0]=20, 3000);
```

Notice the change made in the main thread, is reflected in the worker.



Review the code in the worker that prints data in a loop.

```
console.log("--- Web Worker Ready! ---");

let state
onmessage = (event) => {
  state = event.data;
  iterateAndPrint(); // once data is received, invoke the function to print
};
```

```
// print data in global variable. Iteratively call same function every second.
const iterateAndPrint = () => {
  console.log(state);
  setTimeout(() => iterate10Times(), 1000);
};
```

Browser Support

	Chrome	Edge	Firefox	IE	Opera	Safari
Desktop	60	16	55	No Support	No Support	10.1
Mobile	60	N/A	55	No Support	No Support	No Support

Atomics

One of the problems with `SharedArrayBuffer` is that it's difficult to say when the data will be synchronized among threads or agents. Yes, it will eventually get synchronized but the speed of synchronization is system dependent, i.e. based on availability of resources.

Atomics helps solve this problem about predictability. Atomics provides many static functions that enables a unit work performed. An operation like `store` will be predictable and won't return, till finished.
The following code snippet saves and reads using Atomics' static functions.

```
// store a value 10 at array index 0 for the variable (SharedArrayBuffer) ia
 Atomics.store(ia,0,10);

// return value at zeroth position in the variable ia.
Atomics.load(ia,0);
```

Refer to complete list of static functions at this link https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Atomics

Browser Support

	Chrome	Edge	Firefox	IE	Opera	Safari
Desktop	60	16	55	No Support	No Support	10.1
Mobile	60	N/A	55	No Support	No Support	No Support

Conclusion

ECMA Script is making JavaScript sophisticated with new features and specifications year on year. The release process enables major browsers to adapt new specifications and make it ready to use for developers that desire bleeding-edge web technologies. Remember the table we discussed at the beginning of this article? Well keep a tab on Stage 3 specifications for upcoming features.

Refer to the process documentation in links and references section below for more details.

Links and References

Process documentation and stage definitions for ECMA Script https://tc39.github.io/process-document/

TC39 proposals https://github.com/tc39/proposals

Specification for Object.values and Object.entries https://github.com/tc39/proposal-object-values-entries

Specification for getOwnPropertyDescriptors https://github.com/tc39/proposal-object-getownpropertydescriptors

Finished proposals- TC39 https://github.com/tc39/proposals/blob/master/finished-proposals.md

APIs documentation by Mozilla

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/padStart
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/padEnd
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer

Microsoft® MVP Most Valuable Professional

Keerti Kotaru
Author

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

Thanks to Ravi Kiran for reviewing this article.

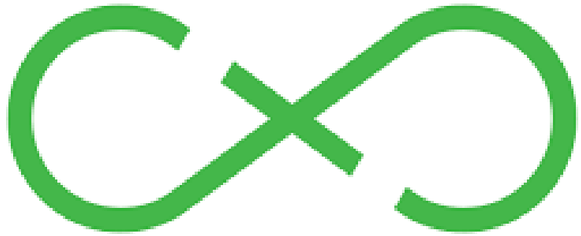
Rahul Sahasrabuddhe

FLUX VS. MVC: COMPARING TWO DESIGN PATTERNS

Do I have to master yet another architectural pattern? I was just getting pro in MVC!

Well, you may have this quandary if you have already heard about the Flux design/architecture pattern from Facebook, and what it does. In this article, we will compare the MVC architecture with Flux from various perspectives.

Let's dig in!



INTRODUCTION

Unless you have been hiding under a rock or you are not a web application developer, I believe you have had some experience (or familiarity) with the **MVC architecture.**

Historically, there has been a lot of MVC patterns, followed across various technologies including ASP.NET, PHP, Ruby on Rails, various JavaScript frameworks, as well as libraries. While it is a popular architecture for web applications, you also have some thick client technologies like WPF follow a variation of MVC a.k.a. the MVVM pattern.

Flux is a new application architecture introduced by Facebook in May 2014. The team at Facebook stated that it was designed by Facebook to overcome the limitations of MVC architecture.

We'll see how!

MVC Popularity

MVC is a popular and widely-used architecture for developing web applications.

MVC based apps span across various verticals like Business, Shopping, Education, Government, Health etc. The MVC architecture is widely popular across different technologies like ASP.NET MVC, PHP, RoR as well as any other frameworks or technologies that use the MVC architecture. In this context, it is fair to suggest that the MVC architecture is the de-facto architecture used in modern web application development, irrespective of the technology.

A Two-minute Primer on MVC

MVC is based on SoC (Separation of Concerns) principle.

It essentially means that you separate out each concern (or responsibility) in such a way that each component addresses a specific concern (or responsibility) in a modular fashion.

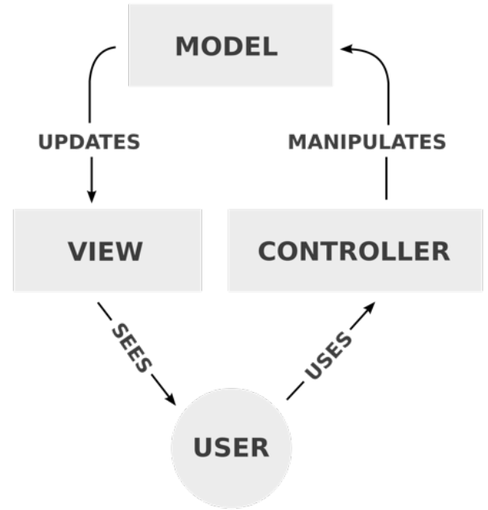


Figure 2: MVC Architecture

Source: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

I am assuming that you must have already seen this diagram umpteen times and have a good understanding of the functioning of MVC architecture. Hence I would not be focusing my efforts on explaining all the aspects depicted in the picture above.

Editorial Note: *If you are new to MVC or ASP.NET MVC, check this* - www.dotnetcurry.com/aspnet-mvc/922/aspnet-mvc-pattern-tutorial-fundamentals

"Patterns are awesome; developers aren't?" – Model View Confusion?

Usually, the concept or architecture is not the culprit; it is the use (or overuse or wrong use) of the architecture to solve business problems, that creates issues.

Let's look at some key aspects of what led to the perception that MVC architecture has problems (please do note the word "perception" here):

Bidirectional vs. unidirectional data flow: MVC design pattern itself does not put any constraint when it comes to flow of direction of data between Model, View and Controller.

In MVC, as you know, the Model is responsible for managing and maintaining the state of data and the controller does the job of data updates based on user interactions in the View. So, the controller maintains application state and also mediates between Model and View.

This gets really difficult to manage in complex applications and hence two-way data binding works well in cases where quicker updates are to be made to the Model and View based on user interactions. In fact, two-way data binding is widely used in many JavaScript frameworks including Angular. So, anybody's claim that bidirectional data flow in MVC is an issue does not make MVC a flawed pattern.

Flux, as an architecture or pattern, now has a built-in structure that mandates unidirectional flow. The argument in favor of unidirectional data flow is that it promotes a clean data flow architecture. This ensures that data flows in your app in a single direction giving you better control over it. It also promotes loose coupling since the state of the application is contained in specific stores.

MVC creates too many Controllers and/or Views: Usually any MVC architecture implementation would have multiple Controllers.

So, if you have a complex application built using MVC, it is evident that you may end up with as many Views and Controllers, as the functionalities you have. And *that* could create a problem when many Views and Controllers interact with each other.

However, the pattern does not mandate you to have one Controller per View.

You can logically group together some controllers and optimize the code structure complexity accordingly. MVC does not mandate any notion of One Model > One View > One Controller structure. As long as each component is managing the SoC as per design principles, you can have logical collection of Controllers, Models and even Views.

For example, if you have a View that displays data from the User Profile and Shopping Cart model, then you can do this using the ViewModel pattern. The ViewModel in this context will be used to compose multiple models into a single one that be consumed by the View. Here is a link that explains some more ways of achieving this.

So, perceiving that the MVC architecture results into a complicated MVC structure, is incorrect.

MVC does not scale: (the scaling myth): It is often mentioned that an MVC based application has issues with scaling.

While an architecture may be partly responsible for scaling related limitations, it is not the only factor. There are many web applications that are developed using the MVC architecture and are proven to be scalable to meet customer demands.

Having said that, information is the most challenging aspect of a software when it comes to scaling. Flux tackles information scaling well, as it puts information first and foremost, above everything else.

Let's get Flux'ed

Before we begin, let's get de-MVC'ed first and look at Flux as a new architecture.

By this, I mean that we should leave all the MV-Confusion out and look at Flux with a clean-slate.

The dictionary definition of the word "flux" is the act/action of flowing or moving. The essence of Flux as an architecture lies in the flow of information in an application.

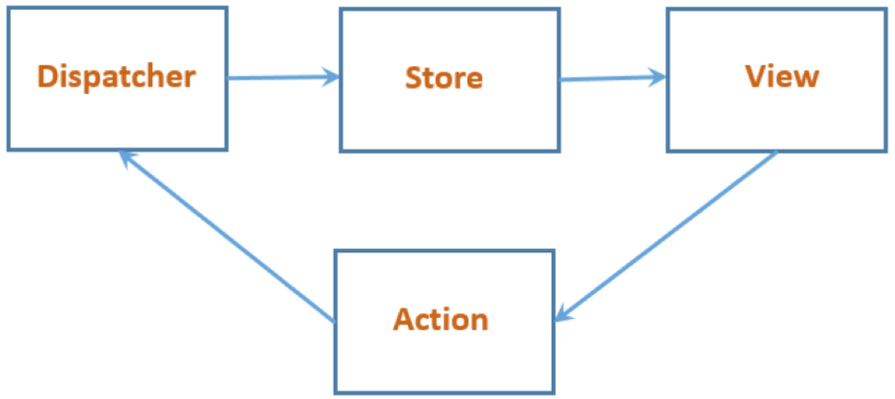


Figure 3: Flux Architecture

The Flux architecture has the following components:

1) Action: Action is raised by the View when the user interacts with the UI controls in the View.

2) Dispatcher: It holds the context to data store (or stores) and propagates the action from View to Store(s). Dispatcher would receive the Action from the View and would notify the Store.

3) Store: Store is registered with Dispatcher. Store contains the data. It receives an update event from Dispatcher and will respond to it. The response would be another "change" event (this is **not** the same event as the dispatcher event) raised to which the View is subscribed.

4) View: View would respond to the change event and would make appropriate changes.

The following diagram explains the architecture through a very simple example of a button being clicked on the View (or a web page) and how the various components of the Flux architecture handle this scenario.

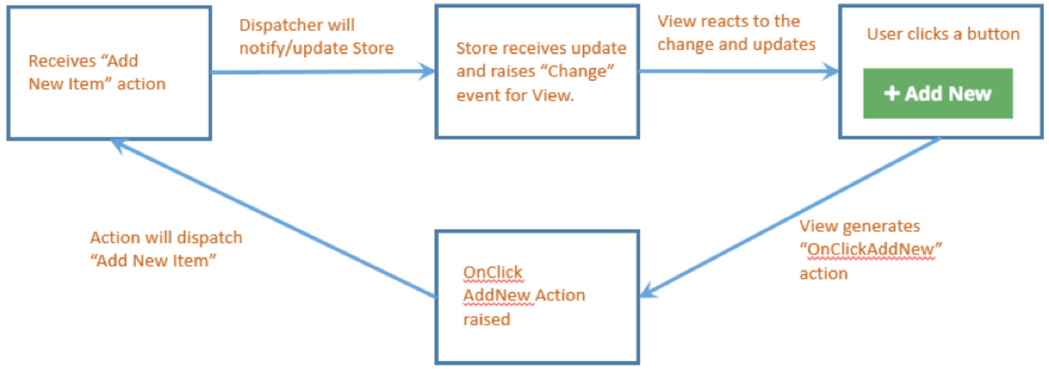


Figure 4 : Flux in the works

Here are some key aspects to remember:

1) The Dispatcher is an event hub for the View. It handles all the Actions and calls the appropriate callbacks registered by Store(s) . So, View has an Action Creator that "creates" Actions. Since the View has frequent interactions with user which would result in a repetitive process of creation of Actions/events that are sent to Dispatcher; Action Creator provides required abstraction to handle this aspect more elegantly. Different flavors of Flux architecture have Dispatcher implemented in different ways.

2) Store needs to register with Dispatcher to get updates on Actions.

3) Dispatcher is more like a router for routing Actions to Store.

4) Store has data and would process it based on Actions received. So, Store is not the same as Model in MVC. It can contain models though. This is also the only component in Flux architecture that knows how to update data. This is where the business logic resides.

5) Store itself would raise events to update View based on Actions.

6) View renders itself again after the change event is raised by Store.

Some important aspects to note in context of MVC pattern:

1) Dispatcher is not the same as Controller. Controller is responsible for the communication between View and Model. Dispatcher is only responsible for handling Actions raised by View in the Flux pattern and inform the Store about the same (so essentially one-way communication). Store, in turn, would raise an event so that View is updated.

2) Controller plays a key role in MVC as it is the hub for essentially managing communications both ways. Due to the very nature of Flux pattern being unidirectional, this flow is managed through following chain of actions: Action > Dispatcher > Store > View.

3) With more Views and Controllers getting added, MVC could create a complicated flow of information. In case of Flux, as the application grows, there will be more Views added – and so will be more Stores. View will continue to raise Action that will be sent to Dispatcher. Dispatcher will then invoke the Store(s) that are registered for that action and that will invoke the right View. That is how Flux is Flex (flexible) also.

In a nutshell, Flux, by design, mandates the separation of concern principles in a more stringent manner through unidirectional flow and hence being more stringent in structure.

Please use this link for a simple demo elaborating the Flex architecture.

FLUX-BASED JAVASCRIPT FRAMEWORKS

There are various Flux-architecture based JavaScript implementations that you can explore and decide to use the right one based on your requirements. Here is a quick table summarizing various Flux-based frameworks:

Framework	Git Repo	Git Stars	Highlights
Facebook's Flux	https://github.com/facebook/flux	14,000+	1) Backed by Facebook. Has got great community support. 2) Multiple Stores per application but each Store is singleton. 3) Dispatcher is singleton.
Reflux	https://github.com/reflux/refluxjs	5,200+	1) A variation of Flux, since it does not have a Singleton Dispatcher. 2) Actions act like Dispatcher. And hence there are no Action Creators. 3) This is termed as a terse implementation of Flux due to absence Dispatcher and Action Creator etc. components.
Redux	https://github.com/reactjs/redux	34000+	1) Store does not contain change logic 2) Store is Singleton for an application 3) No dispatcher implementation. Store has Dispatcher built in.
Alt	https://github.com/goatslacker/alt	3400+	1) Pure Flux implementation 2) An "isomorphic" (i.e. the ability of being able to run same code both on client and server; allowing server-side rendering without changes in code) implementation that binds server-side and client-side JavaScript well (typically in context of Node.JS based full stack development)
Flummox	https://github.com/acdlite/flummox	1600+	1) Relatively new in the overall Flux-based JS Frameworks ecosystem. 2) This is also called as isomorphic JavaScript implementation. 3) Actions and Stores are created first and then they are brought into a Flux class so that you follow a Flux-architecture.

There are quite a few Flux based JS frameworks that you would find. Currently there is a lot of hype and discussion around Flux as a design pattern and hence there are a lot of new JS frameworks following this pattern that are popping up. The action is supposed to continue like this for a while until, as usual, some clear winners emerge in terms of reliability, extendibility and robustness etc.

Facebook's own Flux-based JavaScript framework would, undoubtedly, be the front-runner.

Editorial Note: *To understand the difference between Flux, Reflux and Redux, check this*
https://stackoverflow.com/questions/32461229/why-use-redux-over-facebook-flux/32920459#32920459

https://npmcompare.com/compare/flux,redux,redux-saga,reflux

When Should I consider using Flux-based Design for UI?

Obviously, there is absolutely no need to change or re-architect any of your MVC-based applications.

Don't fix it (or Flux it) if it ain't broken!

For new web application development, you can think of using Flux-based development since it provides a clean design pattern to follow.

However, do keep the following things in mind:

- Learning curve: Given that you are working with a specific design philosophy (Flux), you will need to ensure that you understand the concepts well before writing any Flux-based JavaScript Framework. As I said before, a pattern is as good as how it is implemented.

- UI Complexity: You will need to ascertain if the use of Flux architecture is going to help the cause. Would you rather go with a React JS implementation for a simple functional UI with minimal complexity, or would you need Flux to handle various intense interactions that are mandated by the kind of application you are building.

- Choice of Flux framework: There are quite a few frameworks already. You will need to choose the one that suits well for your needs.

These are of course some factors that you ought to consider. Additionally, it would be useful to get your hands dirty by playing around with readily available samples/demos so that you can get a real taste of usefulness of the Flux-based frameworks (as against MVC-based frameworks).

Seeing is believing!

Conclusion:

Are you flummoxed by Flux yet? Or are you still held up with the Model-View-Confusion?

Well, whichever state you are in, it is evident that Flux architecture based JavaScript framework is another step taken towards building efficient and clean web applications.

It can be looked upon as an attempt of applying the learning from MVC-based JavaScript web development and building something that can withstand ever-growing expectations from various stakeholders for building rich UI web applications.

With the rapid pace at which JavaScript ecosystem has evolved and is evolving, I wouldn't be surprised if there is yet another design pattern (eg: redux or similar) that is just around the corner. So, the flux continues...

Rahul Sahasrabuddhe
Author

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

Thanks to Damir Arh, Yacoub Massad and Suprotim Agarwal for reviewing this article.

OPEN SOURCE FRAMEWORK

Muhammad Ahsan

dynamic queue

SEAMLESS COMMUNICATION WITH MESSAGE QUEUES

DynamicQueue is a highly extensible framework crafted in C# with the good coding practices.

It defines an abstraction to communicate seamlessly with message queues, independent to any message broker implementation (message broker agnostic). The abstraction is based on different communication patterns like producer-consumer, request-response etc., as well as message direction (inbound or outbound).

This framework hides all the implementation details and provides simple interfaces to work with different message brokers, without the big hassle of understanding each message broker and their clients. It also facilitates to switch between different message brokers without changing the code (plug and play fashion).

Message Queues

Whether your application is small, real-time, or enterprise level; message queues play an important role in the smooth execution of the application.

Message queues help us to achieve segregation of different components of the application, running on different machines, environments or maybe geographically distant, with seamless communication between them.

There are many message brokers available in the market from freeware to commercial, offering different features with continuous improvements in every release. And sometimes it is hard to choose between these message brokers.

Furthermore, you will be faced with a few more decisions about which features are best suited for your application, as well as the client library used to integrate your application.

Do not be exhausted, yet! You are just half-way through!

Depending on the message broker, you may also need to tackle serialization/de-serialization, thread safety, exception handling, logging and some other aspects of the application.

The challenges mentioned above, can be easily managed with basic skills and good programming practices.

The real challenge comes into the picture when and if you need to use more than one message broker or switch between message brokers in your application.

Switching requirement scenario could be using free message broker in a development environment and a commercial one, in production. Or there could be scenario that one component of the application is using one type of message broker and the other component is using a different one altogether.

Switching or using more than one message broker within an application is not a trivial task. You need to define some kind of abstraction to reuse the code and also maintain implementation for different message brokers.

DynamicQueue Abstraction

The core of the framework is in its abstraction (based on communication patterns). We can generalize the communication pattern in three major categories:

1. Fire and forget (FaF) [synonyms: produce-consumer, worker queues, push-pull]
2. Request and response (RaR) [synonyms: server-client]
3. Publisher and subscriber (PaS)

There is not a single fatty interface to interact with different categories as it is against the SOLID principles (the letter I – interface segregation). The DynamicQueue framework has defined two interfaces (inbound and outbound) for each category. As of now, the framework handles the first two categories (FaF and RaR) and the following table describes the available interfaces:

Inbound & Outbound Interfaces

Pattern	Inbound-Interface	Outbound-Interface
FaF	IInboundFaFMq<TMessage>	IOutboundFaFMq<TMessage>
RaR	IInboundRaRMq<TRequest, TResponse>	IOutboundRaRMq<TResponse, TRequest>

The Core DLL

All the interfaces, abstract classes and common stuff are available in the core dll (*MessageQueue.Core.dll*).

Configuration

One big aspect for any application or component is configuration. The DynamicQueue framework does not depend on any particular configuration store; rather, it depends only on `IQueueConfigurationProvider` interface available in `MessageQueue.CofigurationProvider.Core` dll.

The interface has only one method with the following signature:

```
Dictionary<string, string> GetConfiguration(string configurationIdentifier);
```

The configuration for any particular interface can be grouped by identifier and the method takes that identifier as an input parameter to return the configuration against that identifier.

There is default implementation for this interface named as `AppSettingsConfigurationProvider` which retrieves configuration from `AppSettings` (stored in app.config or web.config file).

Here is a sample of configuration stored in AppSettings and grouped by '`MyConfiguration`' identifier:

```
<add key=”MyConfiguration:UserName” value=”guest” />
<add key=”MyConfiguration:Password” value=”guest” />
<add key=”MyConfiguration:Address” value=”localhost” />
...
```

Note that ':' is just a separator between the parameter key and the configuration identifier.

Please see the '**Supported Message Brokers & Configuration**' section later in this article for detailed information about available configuration parameters.

Configuration Validation

All configuration parameters are validated before being used. A proper exception with an error message will be thrown in the following scenarios:

- Not supported parameter passed
- Not applicable parameter passed
- Missing required parameter
- Invalid parameter value

Logging

Logging is a major aspect of any application and has been well catered to in the DynamicQueue framework.

Although logging is optional, it is highly recommended to be used in production systems to ease troubleshooting. There is a core logging interface named as `IQueueLogger`. The framework provides a default implementation using `NLog`, but is not limited to that. If you need to use any other or custom logging framework, simply implement the aforementioned logging interface.

The Queue Factory

`MessagingQueueFactory` is the class responsible to create any interface implementation. It exposes static methods which takes three parameters as follow:

1. configurationProvider of type **IQueueConfigurationProvider** [*required*]
2. configurationIdentifier of type **string** [*required*]
3. logger of type **IQueueLogger** [*optional*]

```
public static IOutboundFaFMq<TMessage> CreateOutboundFaF<TMessage>
(IQueueConfigurationProvider configurationProvider, string configurationIdentifier,
IQueueLogger logger = null)
{…}
```

Exception Handling

Writing code which does not crash the application in any scenario is the fundamental task at hand. Defensive coding practice helps a lot to avoid unhandled scenarios and the DynamicQueue framework also practices the same. It throws `QueueException` (*inherits from .Net base Exception class*) in any failure (and logs it if the logging is configured).

Serialization

Depending on the type of the message broker, serialization requirements vary (binary, json etc.).

DynamicQueue manages any kind of serialization and de-serialization in a seamless fashion. All the interfaces take the type of the message (type parameter - T) at the time of creation and use the *Newtonsoft.Json* library for serialization and de-serialization. For binary serialization, the message is converted into a JSON string and then into bytes using UTF8 encoding.

Non-blocking Message Receiver

Message receiving for any communication pattern is non-blocking and event based.
All inbound interfaces expose event delegate and once registered, will be fired up on any message received from the message broker. There is also an asynchronous event delegate for FaF inbound (*may come for other inbound interfaces in future*).

Please see the following example which demonstrates the non-blocking message receiving:

```
*** Initialization Part ***

// Creating FAF inbound interface for ZeroMq (message type is 'string').
var inboundMessageQueue = MessagingQueueFactory.CreateInboundFaF<string>(new
AppSettingsConfigurationProvider(), "ZeroMqFaFInbound", new
NQueueLogger("Default"));

// Registering event handler (will be called whenever message would be available in
queue).
inboundMessageQueue.OnMessageReady += ZeroMqInboundMessageQueue_OnMessageReady;
...
*** Message Handler Part ***

// Message handler
private static void OnMessageReady (string message, IMessageReceiveOptions options)
{
  // message processing goes here…
}
```

Thread Safety

Most of the message brokers' client libraries are thread safe. But in case a library is not thread safe, DynamicQueue makes sure that it can be used concurrently thereby managing thread safety. So again, you will be benefited from this core aspect of managing thread safety and hassle-free switching between the message brokers whenever needed. (*In the current supported message brokers, ZeroMq is the only one which is not thread safe. DynamicQueue uses locking mechanism provided by the .Net to synchronize between multiple threads*)

Supported Message Brokers & Configuration

As of now, DynamicQueue supports three message brokers:

1. ZeroMq
2. RabbitMq
3. Microsoft ServiceBus

Note: *Interfaces implementation has to be exactly the same while configuring DynamicQueue as provided in the following tables (under the heading Interfaces Implementation) for each supported message broker.*

1. ZeroMq

ZeroMq is a light weight, efficient, in-process message broker. Please see the following tables for available interfaces and configuration parameters:

Interfaces Implementation

Pattern	Inbound-Interface	Outbound-Interface
FaF	MessageQueue.ZeroMq.Concrete.Inbound.ZmqInboundFaF`1, MessageQueue.ZeroMq	MessageQueue.ZeroMq.Concrete.Outbound.ZmqOutboundFaF`1, MessageQueue.ZeroMq
RaR	MessageQueue.ZeroMq.Concrete.Inbound.ZmqInboundRaR`2, MessageQueue.ZeroMq	MessageQueue.ZeroMq.Concrete.Outbound.ZmqOutboundRaR`2, MessageQueue.ZeroMq

Configuration

Name	Description	Required
Address	The address of the queue (no server as it is in-memory)	✓
Implementation	The relevant implementation (inbound or outbound)	✓

2. RabbitMq

RabbitMq is a widely used open source message broker. In addition to all the fancy features, it also supports AMQP (*Advanced Message Queuing Protocol*) which is an open standard protocol for message brokers (more on AMQP protocol). Please see the following tables for available interfaces and configuration parameters:

Interfaces Implementation

Pattern	Inbound-Interface	Outbound-Interface
FaF	MessageQueue.RabbitMq.Concrete.Inbound.RmqInboundFaF`1, MessageQueue.RabbitMq	MessageQueue.RabbitMq.Concrete.Outbound.RmqOutboundFaF`1, MessageQueue.RabbitMq
RaR	MessageQueue.RabbitMq.Concrete.Inbound.RmqInboundRaR`2, MessageQueue.RabbitMq	MessageQueue.RabbitMq.Concrete.Outbound.RmqOutboundRaR`2, MessageQueue.RabbitMq

Configuration

Name	Description	Required
Address	The address of the RabbitMq server	✓
Implementation	The relevant implementation (inbound or outbound)	✓
QueueName	The queue name	✓
UserName	Username to connect with server	✓
Password	Password to connect with server	✓
Port	The server port (default will be used if not passed)	×
Acknowledgment	The message acknowledgment setting (inbound only)	×
MaxConcurrentReceiveCallback	The max number of concurrent calls to the receive handler (default 1)	×
ExchangeName	The exchange name	×
RoutingKey	The routing key	×
ConnectionTimeoutInMinutes	The connection timeout in minutes	×

3. Microsoft ServiceBus

ServiceBus is an enterprise level message broker from Microsoft. It can be used either on-premises or cloud based (Azure).

Please see the following tables for available interfaces and configuration parameters:

Interfaces Implementation

Pattern	Inbound-Interface	Outbound-Interface
FaF	MessageQueue.ServiceBus.Concrete.Inbound.SbInboundFaF`1, MessageQueue.ServiceBus	MessageQueue.ServiceBus.Concrete.Outbound.SbOutboundFaF`1, MessageQueue.ServiceBus

Configuration

Name	Description	Required
Address	The address of the ServiceBus server	✓
Implementation	The ServiceBus endpoint address	✓
QueueName	The queue name	✓
Acknowledgment	The message acknowledgment setting (inbound only)	×
MaxConcurrentReceiveCallback	The max number of concurrent calls to the receive handler (default 1)	×

Framework does not create queues except in RaR (request-response) pattern where it needs to create queue for the responses. As ZeroMq is in memory, so no need to create queue before use.

DynamicQueue Example

In this section I will walk you through the complete cycle from configuring, creating, sending and receiving a message using the *DynamicQueue* framework in C#. This Example uses the *FaF* communication pattern with the sender and receiver components, and *RabbitMq* as message broker. So, bear with me and get ready to see the magic of the framework.

Configuration

For this example, the configuration will be stored in the AppSettings (web.config or app.config).

```
<!-- RabbitMq Outbound -->
<add key=”RabbitMqFaFOutbound:UserName” value=”guest” />
<add key=”RabbitMqFaFOutbound:Password” value=”guest” />
<add key=”RabbitMqFaFOutbound:Address” value=”localhost” />
<add key=”RabbitMqFaFOutbound:QueueName” value=”Test_SampleQueue” />
<add key=”RabbitMqFaFOutbound:Implementation”
 value=”MessageQueue.RabbitMq.Concrete.Outbound.RmqOutboundFaF`1, MessageQueue.
RabbitMq” />

<!-- RabbitMq Inbound -->
<add key=”RabbitMqFaFInbound:UserName” value=”guest” />
<add key=”RabbitMqFaFInbound:Password” value=”guest” />
<add key=”RabbitMqFaFInbound:Address” value=”localhost” />
<add key=”RabbitMqFaFInbound:Acknowledgment” value=”true” />
<add key=”RabbitMqFaFInbound:QueueName” value=”Test_SampleQueue” />
<add key=”RabbitMqFaFInbound:Implementation”
  value=”MessageQueue.RabbitMq.Concrete.Inbound.RmqInboundFaF`1, MessageQueue.
RabbitMq” />
```


Note: *A queue with the name `Test_SampleQueue` should be be present.*

Sender

The following code will create the outbound interface for RabbitMq to send the message into the queue. The code uses default configuration provider (`AppSettingsConfigurationProvider`) and the default logger (`NQueueLogger` – logging with NLog library). It will pass the ‘`RabbitMqFaFOutbound`’ to the `create` method as configuration identifier to load the related configuration.

```
class Program
{
  static int Main(string[] args)
  {
    // Creating sender using factory.
    var messageSender = MessagingQueueFactory.CreateOutboundFaF<string>(new
      AppSettingsConfigurationProvider(), “RabbitMqFaFOutbound”, new
      NQueueLogger(“Default”))

    // Sending the message.
    messageSender.SendMessage(message);
  }
}
```

Receiver

The following code will create the inbound interface for RabbitMq to receive messages from the queue. The code uses default configuration provider (`AppSettingsConfigurationProvider`) and the default logger (`NQueueLogger` – logging with NLog library). It will pass the ‘`RabbitMqFaFInbound`’ to the `create` method as the configuration identifier to load the related configuration.

```
class Program
{
  static int Main(string[] args)
  {
    // Creating receiver using factory.
    var messageReceiver = MessagingQueueFactory.CreateInboundFaF<string>(new
      AppSettingsConfigurationProvider(), “RabbitMqFaFInbound”, new
NQueueLogger(“Default”));

    // Registering event handler.
    messageReceiver.OnMessageReady += OnMessageReadyHandler;
  }

  // Message receiver.
  private static void OnMessageReadyHandler(string message, IMessageReceiveOptions
    messageReceiveOptions)
  {
    Console.WriteLine(“Pulled message successfully (RabbitMq)...” + message);

    if (messageReceiveOptions.IsAcknowledgmentConfigured)
    {
      messageReceiveOptions.Acknowledge();
    }
  }
}
```

Nuget Packages

Nuget packages are available for all the supported message brokers. Please go to the nuget.org and search for DynamicQueue to find the packages OR click here to navigate to the page.

Github Repository

DynamicQueue is an open source framework and the source code is available on github public repository at: DynamicQueue Github Repository

Conclusion

The current software market is very competitive and rapid application development is a widely used development model. DynamicQueue framework has many features like on-the-fly switching of message brokers and is one of the key components which helps to deliver efficiency and aids in heavy lifting (for any particular problem).

Off the shelf components are very effective in this kind of environment to increase the productivity and reduce the delivery time of the application ■

Muhammad Ahsan

Author

Muhammad Ahsan is a passionate Software Engineer/Solution Architect with a strong feeling of "Do the Right Thing. Do the Thing Right". He excels in Microsoft's .NET ecosystem with great analytical skills and has more than six years of experience of full stack development (HTML, JavaScript, jQuery, Ajax, ASP.Net MVC, C#, LINQ, Entity Framework, WCF, Web API, Silverlight, MS SQL Server, Windows Workflow Foundation). He is also experienced in analysis, architecting (service oriented) and developing large scale modular applications with state of the art technologies and standards in agile development

Thanks to Damir Arh, Hassan Gulzar, Marco Incalcaterra and Chirag Patel for reviewing this article.

Damir Arh

C# 7.x AND BEYOND

Visual Studio 2017 15.3 update was a big milestone for C#. It was the release vehicle for C# 7.1 – the first minor version of C# 7. This article will take a closer look at its new features and the plans for future versions of the language.

With Visual Studio 2017, Microsoft increased the release cadence for C#.

Between the major versions, which were historically aligned with new Visual Studio versions, they started to release minor versions as part of selected Visual Studio 2017 updates. Minor versions will include smaller new features, which don't require changes to the Common Language Runtime (CLR).

Larger features will still be released with major versions only.

C# 7.1 – WHAT'S NEW

C# 7.1 was released in August 2017 as part of the 15.3 update for Visual Studio 2017. Unlike new language releases in the past, this time the new features are not automatically enabled after updating Visual Studio; neither in existing projects, nor when creating a new project.

If we try to use a new language feature, the resulting build error will suggest upgrading the language version in use.

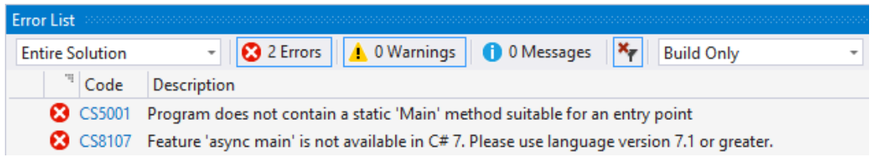


Image 1: Build error for new language featuresThe language version can be changed in the project properties.

On the *Build* tab there is an *Advanced* button, which will open a dialog with a dropdown for selecting the language version. By default, the latest major version is selected, which is 7.0 at the moment.

Editorial Note: *If you are new to C# 7, read our tutorial at http://www.dotnetcurry.com/csharp/1286/csharp-7-new-expected-features*

We can select a specific version instead (7.1 to get the new features) or the latest minor version, which will always automatically use the latest version currently available.

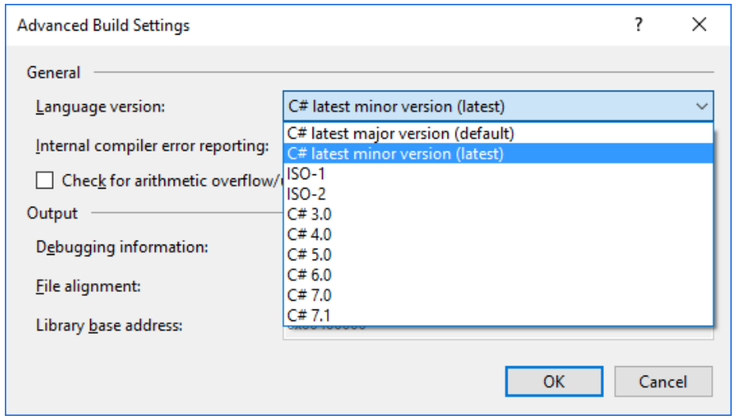


Image 2: Changing the language version

The latter option is not selected by default. This is so that development teams can control how they will adopt new minor language versions. If new language features were automatically available, this would force everyone in the team to update Visual Studio as soon as a single new feature was used for the first time or the code for the project would not compile.

The selected language version is saved in the project file and is not only project specific, but also configuration specific.

Thus when changing the language version in the project properties, make sure you do it for each configuration, or even better: set the *Configuration* on the *Build* tab to *All Configurations* before applying the change. Otherwise you might end up changing the language version for Debug configuration only, causing the build to fail for Release configuration.

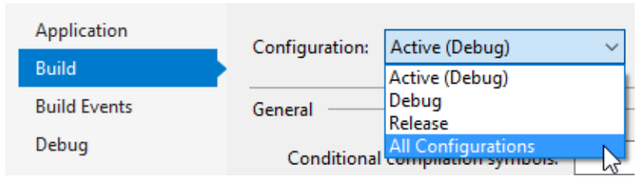


Image 3: Configuration selection on Build tab

For some language features, there is also a code fix available, which will change the language version to 7.1 or to the latest minor version. It will automatically do it for all configurations.

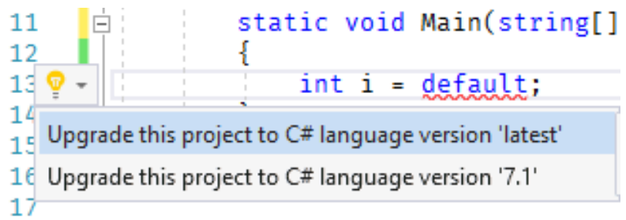


Image 4: Code fix for changing the language version

Four new language features were introduced in C# 7.1.

Async Main

Support for asynchronous `Main` function was already considered for C# 7.0, but was postponed until C# 7.1. The feature simplifies using asynchronous methods with `async` and `await` syntax from console applications. Before C# 7.1, the `Main` method as the program entry point supported the following signatures:

```
public static void Main();
public static int Main();
public static void Main(string[] args);
public static int Main(string[] args);
```

As asynchronous methods can only be awaited when called from inside other asynchronous methods, this required additional boilerplate code to make it work:

```
static void Main(string[] args)
{
  MainAsync(args).GetAwaiter().GetResult();
}

static async Task MainAsync(string[] args)
{
  // asynchronous code
}
```

With C# 7.1, Main method supports additional signatures for asynchronous code:

```
public static Task Main();
public static Task<int> Main();
public static Task Main(string[] args);
public static Task<int> Main(string[] args);
```

When using one of the new signatures, asynchronous methods can be awaited directly inside the `Main` method. The compiler will generate the necessary boilerplate code for them to work.

Default Literal Expressions

Default value expressions can be used to return a default value for a given type:

```
int numeric = default(int);         // = 0
Object reference = default(Object); // = null
DateTime value = default(DateTime); // = new DateTime()
```

They are especially useful in combination with generic types when we don't know in advance what the default value for the given type will be:

```
bool IsDefault<T>(T value)
{
  T defaultValue = default(T);
  if (defaultValue != null)
  {
    return defaultValue.Equals(value);
  }
  else
  {
    return value == null;
  }
}
```

C# 7.1 adds support for default literal expression, which can be used instead of default value expression whenever the type can be inferred from the context:

```
int numeric = default;
Object reference = default;
DateTime value = default;
T defaultValue = default;
```

The new default literal expression is not only useful in variable assignment, it can be used in other situations as well:

- in a return statement,
- as the default value for optional parameters,
- as the argument value when calling a method.

The literal expression syntax is equivalent to the value expression syntax but is terser, especially with long type names.

Inferred Tuple Element Names

Tuples were first introduced in C# 7.0. C# 7.1 is adding only a minor improvement to its behavior. When creating a tuple in C#, element names had to be explicitly given or the elements could only be accessed via default names `Item1`, `Item2` etc.:

```
var coords1 = (x: x, y: y);
var x1 = coords1.x;

var coords2 = (x, y);
var x2 = coords2.Item1; // coords2.x didn't compile
```

In C# 7.1, tuple names can be inferred from the names of variables used to construct the tuple. Hence, the following code now compiles and works as expected:

```
var coords2 = (x, y);
var x2 = coords2.x;
```

Generic Pattern Matching

One of the most important new features in C# 7.0 was pattern matching using the `is` keyword and the `switch` statement. The type pattern allowed us to branch based on the value type:

```
void Attack(IWeapon weapon, IEnemy enemy)
{
  switch (weapon)
  {
    case Sword sword:
      // process sword attack
      break;
    case Bow bow:
      // process bow attack
      break;
  }
}
```

However, this didn't work for generically typed values. For example, the following code didn't compile in C# 7.0:

```
void Attack<T>(T weapon, IEnemy enemy) where T : IWeapon
{
  switch (weapon)
  {
    case Sword sword:
      // process sword attack
      break;
    case Bow bow:
      // process bow attack
      break;
  }
}
```

C# 7.1 extends type patterns to also support generic types, making the code above valid.

C# 7.2

The language development didn't stop with the release of C# 7.1. The team is already working on the next minor version – 7.2. The release date is not yet announced and the new features cannot be tried out easily, although all the specifications and discussions around them are public.

As the release approaches, we can expect that the updated compiler supporting the new features will be included in Visual Studio 2017 Preview, which can safely be installed alongside the current Visual Studio 2017 release.

Several new language features are currently planned for C# 7.2, however they are still subject to change. Some of them could be postponed to a later version and new features could potentially be added as well.

DIGITAL SEPARATOR AFTER BASE SPECIFIER

In C# 7.0, separators were allowed to be used inside numeric literals to increase readability:

```
var dec = 1_000_000;
var hex = 0xff_ff_ff;
var bin = 0b0000_1111;
```

Additionally, C# 7.2 is planned to allow separators after the base specifier:

```
var hex = 0x_ff_ff_ff;
var bin = 0b_0000_1111;
```

Non-trailing Named Arguments

Named arguments were added to C# in version 4. They were primarily the tool to allow optional arguments: some parameters could be skipped when calling a method, but for all the parameters following it, the arguments had to be named so that the compiler could match them:

```
void WriteText(string text, bool bold = false, bool centered = false)
{
  // method implementation
}
// method call
WriteText("Hello world", centered: true);
```

If the parameters are not optional, arguments can still be named to improve code readability and you can even change the order of arguments if you can't remember what it is:

```
WriteText("Hello world", true, true); // difficult to understand
WriteText("Hello world", bold: true, centered: true); // better
WriteText("Hello world", centered: true, bold: true); // different order
```

However, C# doesn't yet allow positional arguments to follow named arguments in the same method call:

```
WriteText("Hello world", bold: true, true); // not allowed
```

According to the current plans, this will become a valid method call in C# 7.2. Positional arguments will be allowed even if they follow a named argument, as long as all the named arguments are still in their correct

position and the names are only used for code clarification purposes.

Private Protected

Common Language Runtime (CLR) supports a class member accessibility level that has no equivalent in the C# language and thus cannot be used: a `protectedAndInternal` member can be accessed from a subclass, but only if the subclass is within the same assembly as the base class declaring the member.

In C#, the base class developer must currently choose between two access modifiers that don't match this behavior exactly:

- `protected` will make the member visible only to subclasses, but they could be in any assembly. There will be no restriction that they have to be placed in the same assembly.

- `internal` will restrict the visibility of the member to the same assembly, but all classes in that assembly will be able to access it, not only the subclasses of the base class declaring it.

For C# 7.2, a new access modifier is planned: `private protected` will match the `protectedAndInternal` accessibility level – members will only be visible to subclasses in the same assembly. This will prove useful to library developers who will not need to choose between exposing protected members outside the library and making internal members available to all classes inside their library.

Conditional Ref Operator

In C# 7.0, support for return values and local variables by reference was introduced. You can learn more about it from my previous article on C# 7.0 in the Dot Net Curry (DNC) magazine.

However, there is currently no way to conditionally bind a variable by reference to a different expression, similar to what the ternary or the conditional operator does when binding by value:

```
var max = a > b ? a : b;
```

Since variable bound by reference cannot be rebound to a different expression, this limitation cannot be worked around with an if statement:

```
ref var max = ref b; // requires initialization
if (a > b)
{
  r = ref a;       // not allowed in C# 7.1
}
```

For some cases the following method could work as a replacement:

```
ref T BindConditionally<T>(bool condition, ref T trueExpression, ref T
falseExpression)
{
  if (condition)
  {
    return ref trueExpression;
  }
  else
  {
    return ref falseExpression;
  }
}
// method call

ref var max = ref BindConditionally(a > b, ref a, ref b);
```

It will however fail if one of the arguments cannot be evaluated when the method is called:

```
ref var firstItem = ref BindConditionally(emptyArray.Length > 0, ref emptyArray[0],
ref nonEmptyArray[0]);
```

This will throw an `IndexOutOfRangeException` because `emptyArray`[0] will still be evaluated.

With the conditional ref operator that's planned for C# 7.2, the described behavior could be achieved. Just like with the existing conditional operator, only the selected alternative would be evaluated:

```
ref var firstItem = ref (emptyArray.Length > 0 ? ref emptyArray[0] : ref
nonEmptyArray[0]);
```

Ref Local Reassignment

There is another extension of local variables and parameters bound by reference planned for C# 7.2 – the ability to rebind them to a different expression. With this change, the workaround for missing conditional ref operator from the previous section would work as well:

```
ref var max = ref b;
if (a > b)
{
  r = ref a;
}
```

Read-only Ref

In performance sensitive applications, structs are often passed by reference to the called function, not because it should be able to modify the values, but to avoid copying of values. There is no way to express that in C# currently, therefore the intention can only be explained in documentation or code comments, which is purely informal and without assurance.

To address this issue, C# 7.2 is planned to include support for read-only parameters passed by reference:

```
static Vector3 Normalize(ref readonly Vector3 value)
{
    // returns a new unit vector from the specified vector
    // signature ensures that input vector cannot be modified
}
```

The syntax is not yet finalized. Instead of `ref readonly`, in could be used. Even both syntaxes might be allowed.

Blittable Types

There is a concept of unmanaged or blittable types in the Common Language Runtime, which have the same representation in managed and unmanaged memory. This allows them to be passed between managed and unmanaged code without a conversion, making them more performant and thus very important in interoperability scenarios.

In C#, structs are currently implicitly blittable if they are composed only of blittable basic types (numerical types and pointers) and other blittable structs. Since there is no way to explicitly mark them as blittable, there is no compile time protection from unintentional changes to these structs, which would make them non-blittable.

Such changes can have a very large impact as any other struct including a struct that became non-blittable, will become non-blittable as well. This can break consumers without the developer being aware of it.

There is a plan for C# 7.2 to add an explicit declaration for blittable structs:

```
blittable struct Point
{
  public int X;
  public int Y;
}
```

The requirements for a struct to be declared as blittable would remain unchanged. However, such a struct would not automatically be considered blittable. To make it blittable, it would have to be explicitly marked with the blittable keyword.

With this change, the compiler could warn the developer when a change to the struct would make it non-blittable while it was still declared as blittable. It would also allow blittable keyword to be used as a constraint for generic types, allowing the implementation of generic helper functions, which require their arguments to be blittable.

C# 8 – WHAT'S NEW

In parallel to the development of the next minor language version, work is also being done on the next major version. All currently planned features are large in scope and impact. They are still in an early prototype phase and likely far away from release.

Nullable Reference Types

This feature was already considered in the early stages of C# 7.0 development, but was postponed until the next major version. Its goal is to help developers avoid unhandled `NullReferenceExceptions`.

The core idea is to allow variable type definitions to contain information, whether they can have a `null` assigned to them or not:

```
IWeapon? canBeNull;
IWeapon cantBeNull;
```

Assigning a `null` value or a potential `null` value to a non-nullable variable would result in a compiler warning (the developer could configure the build to fail in case of such warnings, to be extra safe):

```
canBeNull = null;       // no warning
cantBeNull = null;      // warning
cantBeNull = canBeNull; // warning
```

The problem with such a change is that it breaks existing code: it is assumed that all variables from before the change are non-nullable. To cope with that, static analysis for null safety could be disabled at the project level, as well as at the level of a referenced assembly.

The developer could opt-in to the nullability checking when he is ready to deal with the resulting warnings. Still, this would be in her/his own best interest, as the warnings might reveal potential bugs in his code.

Recursive Patterns

First pattern matching features have been added to C# in version 7.0. There are plans to further extend the support in C# 8.0.

Recursive patterns are one of the planned additions. They would allow parts of data to be matched against sub-patterns.

The proposal lists a symbolic expression simplifier as an example that could be implemented using this feature. It would require support for recursive types as means for representing the expressions:

```
abstract class Expr;
class X() : Expr;
class Const(double Value) : Expr;
class Add(Expr Left, Expr Right) : Expr;
class Mult(Expr Left, Expr Right) : Expr;
class Neg(Expr Value) : Expr;
```

Simplification could then be implemented as a recursive function, heavily relying on pattern matching:

```
Expr Simplify(Expr e)
{
  switch (e) {
    case Mult(Const(0), _): return Const(0);
    case Mult(_, Const(0)): return Const(0);
    case Mult(Const(1), var x): return Simplify(x);
    case Mult(var x, Const(1)): return Simplify(x);
    case Mult(Const(var l), Const(var r)): return Const(l*r);
    case Add(Const(0), var x): return Simplify(x);
    case Add(var x, Const(0)): return Simplify(x);
    case Add(Const(var l), Const(var r)): return Const(l+r);
    case Neg(Const(var k)): return Const(-k);
    default: return e;
  }
}
```

Default Interface Methods

Interfaces in C# are currently not allowed to contain method implementations. They are restricted to method declarations:

```
interface ISample
{
  void M1();                                    // allowed
  void M2() => Console.WriteLine("ISample.M2"); // not allowed
}
```

To achieve similar functionality, abstract classes can be used instead:

```
abstract class SampleBase
{
  public abstract void M1();
  public void M2() => Console.WriteLine("SampleBase.M2");
}
```

In spite of that, there are plans to add support for default interface methods to C# 8.0, i.e. method implementations using the syntax suggested in the first example above. This would allow scenarios not supported by abstract classes.

A library author could **extend an existing interface** with a default interface method implementation, instead of with a method declaration.

This would have the benefit of not breaking existing classes, which implemented the old version of the interface. If they didn't implement the new method, they could still use the default interface method implementation. When they wanted to change that behavior, they could override it, but no code change would be required just because the interface was extended.

Since multiple inheritance is not allowed, a class can only derive from a single base abstract class.

In contrast to that limitation, a class can implement multiple interfaces. If these interfaces implement default interface methods, this effectively allows classes to **compose behavior from multiple different interfaces** – the concept is known as trait and is already available in many programming languages.

Unlike multiple inheritance, it avoids the so called diamond problem of ambiguity when a method with the same name is defined in multiple interfaces. To achieve that, C# 8.0 will require each class and interface to have a most specific override for each inherited member.

When a member with the same name is inherited from multiple interfaces, one override is more specific than the other when its interface is derived from the other one. When neither interface directly or indirectly inherits from the other interface, the developer will need to specify the override he wants to use or write his own override.

By doing so, he will explicitly resolve the ambiguity.

Conclusion:

C# compiler is delivering on the promise of Roslyn: **faster introduction of new features** thanks to a completely new codebase.

At the same time, new features are not forced onto larger teams who prefer to have stricter control over the language version they are using. They can evaluate new features and decide at their own pace when they want to adopt them.

In accordance to the open source model, even the state of upcoming features in future versions of the language is public and available for all to explore or even contribute their opinion to. It's important to keep in mind though, that these features are still work in progress and as such they could change without warning or even be postponed to a later version ■

Microsoft® MVP Most Valuable Professional

Damir Arh
Author

Damir Arh has many years of experience with Microsoft development tools; both in complex enterprise software projects and modern cross-platform mobile applications. In his drive towards better development processes, he is a proponent of test driven development, continuous integration and continuous deployment. He shares his knowledge by speaking at local user groups and conferences, blogging, and answering questions on Stack Overflow. He is an awarded Microsoft MVP for .NET since 2012.

Thanks to Yacoub Massad for reviewing this article.