Issue 26 | Sep-Oct 2016

DNCMagazine

www.dotnetcurry.com

Aspect Oriented Programming with SOLID

C# Async and Await Best Practices

LOG ANALYTICS (OMS) in Microsoft Azure

SCALABLE APPLICATIONS

in Azure
A Conceptual Overview

A Vanity Look at
ASP.NET CORE

.NET CORE

.NET of Future
or
Future of .NET

BROADCAST REAL-TIME NOTIFICATIONS

using SignalR, Knockout JS and SqlTableDependency

THE TEAM

Editor In Chief
Suprotim Agarwal
suprotimagarwal@a2zknowledgevisuals.com

Art Director
Minal Agarwal

Contributing Authors
Christian Del Bianco
Damir Arh
Daniel Jimenez Garcia
Edin Kapic
Rahul Sahasrabuddhe
Vikram Pendse
Yacoub Massad

Technical Reviewers
Damir Arh
Rahul Sahasrabuddhe
Suprotim Agarwal
Yacoub Massad

Next Edition
November 2016

www.dotnetcurry.com/magazine

CONTENTS

EDITORIAL

The response for the 4th Anniversary edition was fansatic! We got tons of feedback and over 100K developers downloaded the magazine. Thank you all for making it worthwhile!

For this edition, we have a bouquet of exclusive articles for you covering .NET Core, ASP.NET Core, C# Async Await Best practices, SignalR, AOP Programming, Scalable Apps and OMS in Azure.

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

Editor in Chief
Suprotim Agarwal

Aspect Oriented Programming with SOLID

Yacoub Massad

This article discusses some options for doing Aspect Oriented Programming (AOP) when we follow the SOLID principles.

Introduction

Aspect Oriented Programming (AOP) tries to solve the problem of code duplication and code tangling that developers encounter while addressing cross cutting concerns. Consider the following code example that tries to address the logging concern for a class:

```
public class DocumentSource :
IDocumentSource
{
 //..

 public Document[]
GetDocuments(string      format)
 {
  try
  {
   using (var context =
         CreateEFContext())
   {
    var documents =
    context
     .Documents
     .Where(c => c.Name.
      EndsWith("." + format))
     .ToArray();
     logger.LogSuccess(
     "Obtained " + documents.
     Length + " documents of
     type " + format +
     Environment.NewLine +
     "Connection String: " +
      connectionString);

     return documents;
    }
   }
   catch (Exception ex)
    {
     logger.LogError(
      "Error obtaining documents of type
" + format +
      Environment.NewLine +
       "Connection String: " +
connectionString, ex);

     throw;
    }
  }
  //..
}
```

The main purpose of this method is to read documents from some database. It also logs success and failure information.

Here is how the method would look like without logging:

```
public Document[] GetDocuments(string format)
{
  using (var context =
CreateEFContext())
  {
    return
      context
        .Documents
        .Where(c => c.Name.EndsWith("."
        + format))
        .ToArray();
   }
}
```

Clearly, the logging code has made the original method less readable. It has *tangled* the real method code with logging code.
This is also a violation of the Single Responsibility Principle.

Also, we expect to find the same logging pattern in many methods all over the code base. Basically, we expect to find the following pattern:

```
try
{
  //Do something here

  logger.LogSuccess(…
  //..
}

catch (Exception ex)
{
  logger.LogError(…
  throw;
}
```

Aspect Oriented Programming allows us to keep the original code as-is, and provides us a way of *weaving* an aspect (e.g. logging) into the code. An aspect is supposed to be defined in a generic way so that it can be applied to many objects in the application. If we take logging as an example, this means that the pattern shown above will only be maintained in a single unit of code (e.g. class) called an aspect, and then applied to many code units.

SOLID is a set of principles that intends to make code more maintainable. If we apply such principles, we will end up with a lot of small classes that each does a single thing well. Also, classes become highly composable, context independent, and thus reusable.

Editorial Note: To learn more about SOLID, please refer to our articles on SOLID principles.

If the `DocumentSource` class, for example, is to be used in many contexts, i.e. many instances of this class are to be used each with different configuration, logging cannot be hardcoded into the class itself. Each instance might want to be logged in a different way. For example, one instance might obtain documents from an on-premises database, while another instance might obtain documents from a database on the cloud. Although such context information is not known by the `DocumentSource` class, it needs to be logged to the log file.

As an additional example of a challenge made by context independence for AOP, consider the example of a retry aspect. A retry aspect can be applied to any object to make some method retry execution on failure. Because different instances are in different contexts, they might require different number of retries before they give up. We cannot, therefore, hardcode the number of retries in the class itself.

This article will discuss some options for doing AOP, and will also show how to deal with context independence.

I have created a repository on GitHub that contains the sample code for the ideas discussed in this article. You can view this repository at bit.ly/dncm26-aop

AOP via decorators

One way to solve the problems described in the previous example is to put the logging logic into a decorator. Here is an example:

```
public class LoggingAwareDocumentSource
: IDocumentSource
{
  private readonly IDocumentSource
  decoratedDocumentSource;
  //..
  public Document[] GetDocuments(string
  format)
  {
    try
    {
      var documents =
      decoratedDocumentSource.
      GetDocuments(format);

      logger.LogSuccess(
      "Obtained " + documents.Length + "
      documents of type " + format);

      return documents;
    }
    catch (Exception ex)
```

```
{
      logger.LogError(
      "Error obtaining documents of type
      " + format, ex);

      throw;
    }
  }
}
```

We then remove the logging code from the original class.

Then in the Composition Root, we decorate the `DocumentSource` object in the following way:

```
var source =
  new LoggingAwareDocumentSource (
    new DocumentSource
    (connectionString),
      new Logger(…));
```

If you want to learn more about how to compose objects in the Composition Root, see the Clean Composition Roots with Pure Dependency Injection (DI) article.

The solution provided here has two problems.

The First Problem: Specificity

The first problem is that this decorator works only for `IDocumentSource`. If we want to do the same thing for another interface, say `IEmailSender`, we need to create another decorator. In other words, each decorator is specific to a single interface.

One way to fix this problem is to minimize the number of interfaces in the application to a few (e.g. two or three interfaces).

For example, we can create two generic interfaces, one for queries, and another one for commands:

```
public interface IQueryHandler<TQuery,
TResult>
{
  TResult Handle(TQuery query);
}
public interface
ICommandHandler<TCommand>
{
  void Handle(TCommand command);
}
```

..and then make all (or most) classes implement one of these interfaces. Of course, for each class, TCommand, TQuery, or TResult might be different. For example, our DocumentSource class would look something like this:

```
public class DocumentSource :
IQueryHandler<GetDocumentsQuery,
GetDocumentsResult>
{
 //..
 public GetDocumentsResult
 Handle(GetDocumentsQuery query)
 {
  using (var context =
  CreateEFContext())
  {
   return
    new GetDocumentsResult(
     context
      .Documents
      .Where(c => c.Name.
       EndsWith("." + query.Format))
       .ToArray());
  }
  //..
}
```

..and the logging decorator can be generic as shown here:

```
public class
LoggingAwareQueryHandler<TQuery,
TResult> : IQueryHandler<TQuery,TResult>
{
 private readonly IQueryHandler<TQuery,
 TResult> decoratedHandler;
 //..

 public TResult Handle(TQuery query)
 {
  try
  {
   var result = decoratedHandler.
   Handle(query);

   logger.LogSuccess(...);

   return result;
  }
  catch (Exception ex)
  {
   logger.LogError(..., ex);
   throw;
  }
 }
}
```

Note: I learned about this idea of separating classes into command handlers and query handlers from the .NET Junkie blog. More specifically, from these two articles: https://cuttingedge.it/blogs/steven/pivot/entry.php?id=91 and https://cuttingedge.it/blogs/steven/pivot/entry.php?id=92. You might want to read them for more information.

Note that this handler can decorate any `IQueryHandler`, regardless of TQuery and TResult.

This is what makes it work as an aspect.
We also need to create another decorator for `ICommandHandler`. So for each aspect, we have two decorators, which is somewhat acceptable.

Although this solution gives us a lot of flexibility, one could argue that such approach makes the code a bit verbose. For example, the following code:

```
var result = parser.Parse(document);
```

will look something like this:

```
var result = parser.Handle(new
ParseQuery(document));
```

Also, the following code:

```
private readonly IDocumentParser parser;
```

will look something like this:

```
private readonly
IQueryHandler<ParseQuery,ParseResult>
parser;
```

Like most things, this approach has its advantages and disadvantages.

The Second Problem: Missing details

If you look carefully at the logging messages that we output via the first decorator (the `LoggingAwareDocumentSource` class), the "connectionString" of the database is not logged.

And in the case where we used the QueryHandler approach, even the number of documents returned is not logged.

This problem, however, is not specific to the use of decorators, or the idea of splitting the classes into query handlers or command handlers. This problem exists because of the generic nature of the logging aspect. Any generic solution has this problem, and there are solutions. Mainly, we need to tell the aspect how to obtain these details. One way to do this is to annotate specific parameters/members with attributes. Consider the `GetDocumentsResult` class that we used as the return value type for our `DocumentSource` class:

```
public class GetDocumentsResult
{
  [LogCount("Number of Documents")]
  public Document[] Documents { get;
  private set; }

  public GetDocumentsResult(Document[]
  documents)
  {
    Documents = documents;
  }
}
```

Notice how we have annotated the Documents property with a custom LogCount attribute. This attribute is defined like this:

```
public class LogCountAttribute :
Attribute
{
  public string Name { get; private set;
}

  public LogCountAttribute(string name)
  {
    Name = name;
  }
}
```

There is nothing special about this attribute. It only holds a "Name" property which will be used as a description when we log the annotated property. The magic should happen in the logging aspect. In our previous example, this is the `LoggingAwareQueryHandler` decorator. This decorator should use Reflection to examine the properties of both the TQuery object and the TResult object, and look for these special attributes.

If they are found, depending on the attribute, it should read some data and include it in the log. For example, we should expect to find something like "...Number of Documents: 120..." in the log file.

We can have several other attributes to annotate different kinds of things. For example, we can create the obvious *Log* attribute to annotate properties of which value we want to log.

Take a look at the source code of the GetDocumentsQuery class and see how the Format property is annotated with the Log attribute. This will make the aspect log the value of this property.

Take a look at the source code of the LoggingAwareQueryHandler class. It has two dependencies on an interface called IObjectLoggingDataExtractor. This interface represents an abstraction of a unit that can extract logging information from some object. One extractor dependency will extract logging data from the query object, and another extractor dependency will extract logging data from the result object. This makes the logging aspect extensible. We can easily add more logging attributes and provide different implementations of this interface to extract logging data based on these attributes. Take a look at the different implementations of this interface in the AOP.Common project.

You can run the Application.CQS application to see the results. Currently the application is configured to log to the console.

Logging context specific information

We still didn't log the "connectionString" of the database. Additionally, we might want to log context information that the `DocumentSource` class itself does not know about. For example, one `DocumentSource` instance might represent documents from department A, and another instance might represent documents from department B. Such information is only known to the application itself, not individual classes. The Composition Root is the entity that represents the application and has all the knowledge about the application. It is the entity that needs to know which instance represents what. In other words, it is the entity that knows the context for each object in the object graph.

Take a look at the Composition Root of the Application.CQS application, i.e. the Program class. It contains the following code:

```
var queryHandler =
 new
 FakeDocumentSource("connectionString1")
  .AsLoggable(
   new LoggingData { Name =
   "connectionString", Value =
   "connectionString1" },
   new LoggingData { Name =
   "Department", Value = "A" });
```

This code creates an instance of the FakeDocumentSource class. This class is like the `DocumentSource` class described in this article, but it returns fake data instead of data from a real database. I did this to make it easier for readers to run the application without a real database.

Notice the call to the AsLoggable extension method. This method is defined in the QueryHandlerExtensionMethods class. It takes an array of `LoggingData` data objects and uses one implementation of the `IObjectLoggingDataExtractor` interface, i.e. the ConstantObjectLoggingDataExtractor class, to force the logging of some constant logging data. In our particular case, we want to log the connection string and the department information.

Metadata classes

One could argue that we are still coupling the logging logic with the original class since the attributes are annotating members of the class itself. There could be a situation where in one instance we want to log the number of documents, and in another instance we don't want to. One solution to this problem is to move the annotations to a special class or classes, and inform the logging aspect about it. For example, we could create a class called `GetDocumentsResultMetaData1` that contains the same properties as the `GetDocumentsResult` class and annotate the properties in `GetDocumentsResultMetaData1` with the logging attributes that we want for a particular context. Then we tell the logging aspect about this class. If we need to log differently in a different context, we create another metadata class, and so on.

The details of this method is beyond the scope of this article, but could be the subject of a future one.

AOP via dynamic proxies

Let's go back to the first problem that we discussed; the specificity problem. We solved it previously by architecting the system in such a way that we have only a few interfaces. Although it is a valid approach, we might decide to architect our system to have more specific interfaces. How can we solve the specificity problem in this case? We can use **dynamic proxies**.

A dynamic proxy (in this context) is a class generated at runtime that can be used to wrap an object to apply some aspects to it.

So instead of creating some hundred decorators at design time, we create one generic aspect, and then use it to generate the hundred decorators at runtime.

Take a look at the FakeDocumentSource class in the Application.CastleDynamicProxy project in the source code. This class is annotated with some attributes. As we have done before, these attributes will be used by the aspect to extract data for logging.

Take a look at the LoggingAspect class. It implements the `IInterceptor` interface from the Castle DynamicProxy library. This interface has one method; the Intercept method:

```
void Intercept(IInvocation invocation);
```

This method allows us to intercept calls to our objects, e.g. the `FakeDocumentSource` class. We can use the invocation parameter to access the *invocation* context. This context includes information such as the intercepted method arguments, the return value of the method, the method metadata, etc. We can use it, for example, to get the method description from the MethodDescriptionAttribute that we used to annotate the `GetDocuments` method in the `FakeDocumentSource` class.

The `IInvocation` interface also has a "Proceed" method that is used to control when to invoke the original method.

In the `LoggingAspect` class, we first collect information about the method itself, such as the method description from the `MethodDescription` attribute. Then we collect logging data from the method arguments, and then finally invoke the original method. If an exception is thrown, we log the logging data we obtained so far along with the exception itself. If no exception is thrown, we extract logging data from the return value and record all logging data.

DynamicProxy works by using the Reflection API in the .Net framework to generate new classes at runtime. For an example on how this can be done, take a look at the example provided by Microsoft in the documentation of the TypeBuilder class.

Logging context specific information

We can log context information as we did before. There is no difference. Take a look at the Composition Root of the Application.CastleDynamicProxy application and the AsLoggable extension method for more details.

A note about Reflection and Performance

We have relied on Reflection to extract logging information. Basically, we searched for parameters or properties that are annotated with specific attributes, and then read their values. This takes more time than directly accessing parameters or properties.
For example, it is faster to run this code:

```
var value = query.Format;
```

than it is to run this code:

```
var value = query.GetType().
GetProperty("Format").GetValue(query);
```

If we log events frequently, the performance cost would be significant.

To fix this problem, instead of making the proxies search for the attributes and extract data via Reflection every time they are invoked, we search for the attributes during the process of creating the proxies, and generate the proxies in such a way that they contain code that directly access the correct pieces of data (properties/parameters).

The details of this approach is beyond the scope of this article, but we may cover this subject in a future one.

AOP via compile-time weaving

Another method for doing AOP is via compile-time weaving. As we did earlier, we annotate our classes/methods/properties/etc. with certain attributes. But this time, a post-compilation process starts, searches for these attributes, and adds extra IL code to our methods inside the compiled assemblies to make the aspects work.

One of the popular compile-time AOP tools in .NET is PostSharp. It contains a lot of built-in aspects such as the logging aspect, and can be used to create our own aspect. Take a look at the many examples provided in the PostSharp website at http://samples.postsharp.net/

One thing to note here is that aspects are applied to classes, not objects. As far as I know, there is no easy way to configure aspects to work differently for different instances.

AOP via T4

Text Template Transformation Toolkit (T4) is a template based text generation framework from Microsoft. We can use this tool to generate code at design-time.

Basically, we can create a T4 template that represents an aspect, and then use it to generate many classes that each represent the customized application of that aspect to a specific unit. For example, we can create a T4 template for a logging aspect and use it to generate fifty decorators (at design time) that each represent a logging decorator that knows how to log information for the class/interface that it decorates. This is basically the same as AOP via decorators, but now the decorators are generated automatically from a template at design time.

The details of using this method for AOP is outside the scope of this article.

For more information about T4, take a look at the documentation form MSDN: https://msdn.microsoft.com/en-us/library/bb126478.aspx

Conclusion:

AOP allows us to remove code duplication and enhances code readability by moving code related to cross-cutting concerns into a single place where it can be easily maintained. This article provided some options for doing AOP. It also discussed the challenge posed by context-independence in SOLID code base, and provided a solution for such a challenge ■

Download the entire source code from GitHub at bit.ly/dncm26-aop

About the Author

Yacoub Massad

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

Scalable Applications in Azure

A Conceptual Overview

Edin Kapic

As a developer who writes web applications, I have witnessed the growth of cloud computing, and specifically Microsoft Azure, with great interest. It can't be denied that cloud applications are potentially much easier to scale and perform because the cloud fabric is elastic in a way, that on-premises environment is not. However, scalability doesn't come by default in your application. If you deploy a simple web application in Azure, it doesn't magically become more scalable. In this article, I will explain why and how to build for scalability, mentioning Azure components that are useful in that effort.

Why Scalability Matters?

If we look up a dictionary, scalability is described as **scale-ability**, which means the ability to scale or to accommodate a growing amount of work in a graceful way. This will be our goal throughout this article.

It is important to highlight that "a graceful way" can mean many things for many applications, but usually it refers to smooth and progressive degradation of service that allows application to work, but with somewhat increased response times. Between returning an error to the user and making the user wait a little more, we should always choo sse the later.

The process of building a scalable application begins with the design of the application. According to scalability experts Martin Abbot and Michael Fisher, who wrote an excellent little book called "Scalability Rules", scalability should be designed for in three phases: the design itself, the implementation and the deployment.

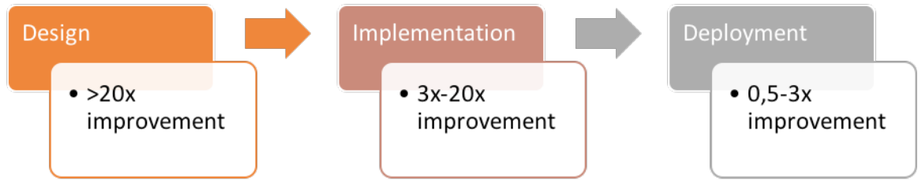


Figure 1: The scalability has to be built for, beginning with the application design

When we **design** for scalability in the first place, we can address many potential issues without compromising existing code base, because it's not there yet. On paper, we can quickly iterate candidate architectures until we have found one that might be the right one. You should be able to identify architectures that can give you at least 20 times your desired workload in this phase.

In the next phase, **implementation**, you build your solution according to the architecture designed in the previous phase. Here, our goal is to scale in between 3 to 20 times our envisioned workload, using techniques that I will outline in this article. Here, the cost of implementing changes is high as there is already an existing codebase to take into account.

Once the application is deployed, we can also streamline the **deployment** architecture to optimize for scalability, but we can't strive to get too much of it. By using a sound deployment architecture, we can get from 50% to 300% more scalability than the one we targeted for initially. Also, leveraging the scale up and down by provisioning extra instances during high load, and shutting them down when the load is reduced, is very important in order to maximize the use of our resources (and money).

If scalability is so good and great, why don't we always build for scalability? Well, that's a good question. The answer lies in the **cost versus benefit** equation. Scalable applications are usually more complex and more expensive to build than an application with no explicit high scalability in mind. So we must ensure that when we build an application, the benefits of adding complexity and additional cost to ensure high scalability are worth it.

Enemies of Scalability

There are features of web applications that make it difficult to scale past their intrinsic user loads. Some of them are introduced by a design choice, but many of them are just "default" features of a normal-looking web application. I will briefly mention four of them.

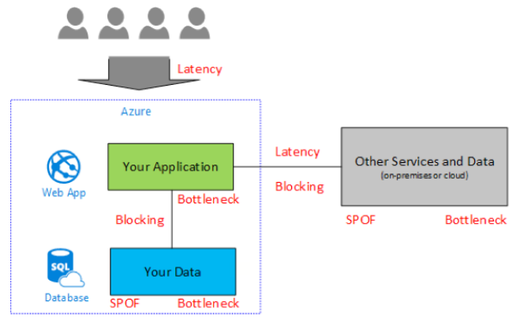


Figure 2: A typical Azure web application with the "enemies of scale" highlighted in red

Bottlenecks

A bottleneck is a single component in the application where all the communication must pass through. Bottlenecks are easy to create, because they are natural points of communication between the layers in our architecture. For example, usually there is a Data Access Layer of some kind in almost any application. If this layer is centralized, all the communication to and from the database passes through it, and we have a bottleneck.

The bottlenecks put a limit to the communication flow in the application, as it can't accommodate more requests and calls than the bottleneck allows for.

Roundtrips and Latency

Unnecessary round trips and communication in our application are also an enemy of *scal*. When our application runs in a local network, the communication is very quick as the latency introduced by the network is almost negligible. But when our application runs in the cloud, and is accessed by thousands of users, the latency increases as there are many network components between the communicating components. The clearest cases are requests from the browser, or the client side to the server side of the application. Each request has to travel across the Internet to the datacenter, passing through routers, switches and other network devices, all of which introduce some delay or latency.

Blocking calls

A web application is by nature an example of request and response model. One of the pages in the application issues a request to the back-end, and the back-end responds. During the request and until the response is issued, the calling page and its executing thread are blocked, but they aren't doing anything useful. They just sit idle waiting for the response to come. It limits our application maximum throughput, as there is a limited number of threads in the web server to make the requests to the back-end. If all of them are blocked waiting for the response, our application will put any additional request on hold until one of these threads is free, which adds to the average response time as well. If the wait is too long, we will see timeouts in our application.

Single Points of Failure (SPOF)

Finally, almost every component of our application can act as a single point of failure. A single point of failure or SPOF is a component that if down, breaks the application. In a single node deployment with no load balancing, the entire application is a huge single point of failure. We can spin more nodes with our application, and it will usually just shift the failure point to the database, as usually all the nodes will share a common database. We must also make the database redundant to avoid a single point of failure.

Components of Highly Scalable Applications

Now that I have shared with you the enemies of scale, it's time to see their heroic counterparts. These are the components of highly scalable applications, architectural patterns and techniques that aid in the pursuit of great scalability.

Minimizing Locks in Storage

In most of the deployments there is a centralized storage location, such as a database, where the "single source of the truth" is stored. The applications read and write data from and to this centralized location, and the dynamics of these actions cause locks to prevent incoherent data. We can't go faster than our database throughput, can we?

Yes, we can! But before doing so, we must prune our storage operations to avoid locking, waits and bottlenecks. By having a streamlined storage layer, we can build our application layers on solid ground.

Relational databases suffer from having to be used for reads and writes at the same time. Read operations need fast response time, while write operations need high throughput. To shield read operations from non-confirmed write operations (and keep the data consistent), relational databases have varying degrees of isolation levels. Higher the isolation level, more the locks and waits in the database operations and lesser concurrency, but with higher data consistency.

The techniques of minimizing storage locking are multiple. We can split or partition the storage into different parts, and then we get more throughput as there are more components in parallel that sum the throughput of all the individual partitions. We can split our database into read and write parts (called Command-Query Responsibility Segregation or CQRS) to eliminate the locking due to transaction isolation levels of our database. We can even dispense with a relational database and use a NoSQL database or even an in-memory database. This is called Polyglot Persistence and it means to choose the best storage mechanism for every data storage need in our application, instead of making a compromise. In Azure we can use Table Storage, SQL Azure, DocumentDB or any IaaS storage technology that's available.

We can also dispense with the immediate consistency of data as in relational databases, and embrace **eventual consistency** that ensures that the data will be eventually updated to the latest version, just not immediately. Almost every scalable architecture uses eventual consistency in one way or another.

Caching

The second component of scalable applications is caching. Caching is undoubtedly the cheapest way to reduce unnecessary roundtrips and latency, by copying the data read from the server, and storing them locally. By using caching we basically trade increased RAM memory consumption for lower response times. All caching mechanisms are prone to stale data, or data that is not reflecting the latest state. Stale data is the inevitable consequence of caching, and can be mitigated at least in part by using multiple caching levels and distributing cache across different servers.

In Azure, the recommended caching solution is Redis. It is a very performant key-value dedicated cache.

Non-Blocking Asynchronous Requests

The third superhero of highly scalable apps is the use of asynchronous, or non-blocking requests. We have seen that an application thread issuing a request is essentially blocked until the response is returned from the back-end. If the programming language allows us to use asynchronous calls, and C# certainly does, we can free the thread from idly waiting for the response. It can have an enormous impact on the throughput of the application, as we are using the existing threads in a much more efficient manner. The same threads can be reused again, while still waiting for the existing requests to the back-end, to respond.

Luckily, using .NET 4.5 or .NET Core with its async/await keywords, and Azure API that has asynchronous methods, helps us accommodate many asynchronous calls in our application, thereby minimizing the useless blocking of the calling threads on the web server.

Queues

With asynchronous calls, we optimize the use of threads, but at the end we still wait for the response to arrive. By using queues, we can decouple from the request-response model and just send a message to the queue, and return immediately. The sender is then free to attend to other requests. The receiving part is also free to get the messages from the queue and process them at its own pace. It complicates things a little bit, but brings together a whole load of improvements. We can offload long operations to be processed by other nodes of the application and then their results are sent back through another queue. We also get fault tolerance as the queue mechanism usually allows for multiple retries if a processing node has failed. We can throttle the requests to the back-end and minimize the pressure on it, which comes in handy to prevent Distributed Denial of Service (DDOS) attacks.

With Azure we have two main queuing mechanisms: Storage Queues and Service Bus queues. You can think of them as the fast, low-level queues; and transactional, higher-level queues, respectively.

Redundancy

The solution to the single point of failure is applying redundancy to all levels of our solution. Redundancy means having more than one instance of a single component in our application. Strictly speaking, redundancy helps to attain high availability, which is the capability of the system to withstand the loss of a component such as a service or a node that crashes. However, as scalable

applications usually experience heavy user loads, it makes their components more prone to crashing, and that's why redundancy is so closely related to scalability.

Redundancy also introduces a whole range of problems such as detecting dead nodes, resolving the request routing to the appropriate healthy node (load balancing, such as Azure Traffic Manager) and having to deal with repeated operations. The atomic operations in our application have to be idempotent, which means that even if they are repeated many times, the end result will be the same. Usually, this idempotency is achieved with the addition of unique operation ID such as GUID, where we can check if a single operation has already been executed before.

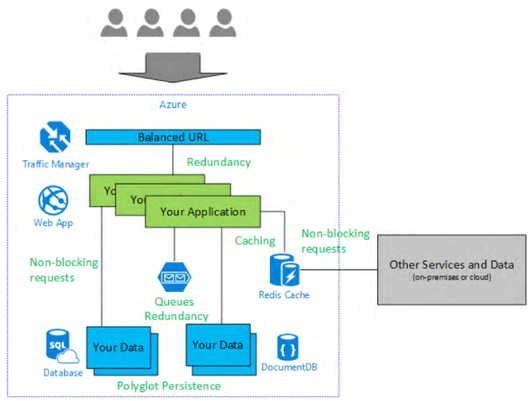


Figure 3: The same application as before, with the components of scalability highlighted in green

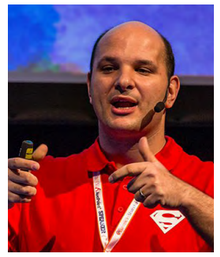
Conclusion

In this article I hope you are convinced that scalability can't be an afterthought. On the contrary, it has to be designed from the very beginning if we really want something that's going to endure high user load.

Scalability adds complexity to our solution. There's no way out of it, but now that we know the enemies of scale and the components of scalability, we are armed with the knowledge to tame this complexity.

If you are interested in an in-depth explanation of how scalable applications are done, with a complete makeover of a web application in Azure, you can check my Pluralsight course that is included in the MSDN subscription ■

About the Author



edin kapic

Edin Kapić is a SharePoint Server MVP since 2013 and works as a SharePoint Architect in Barcelona, Spain. He speaks about SharePoint and Office 365 regularly and maintains a blog www.edinkapic.com *and Twitter @ekapic.*

Microsoft® Most Valuable Professional

Rahul Sahasrabuddhe

.NET Core:

.NET of Future or Future of .NET?

Introduction

.NET Framework beta was released in early 2000s. After its launch, Microsoft realized the need to have a smaller/ compact version of the framework to run on Smartphones/PDAs that used to run Windows Phone operating systems at that time. Hence they came up with .NET CF (Compact Framework). Since then, over a period of time, Microsoft has been refactoring the .NET Framework for various needs & purposes including web development (ASP.NET MVC), RIA (Silverlight), Windows Phone development, and the latest, Windows Store Apps. At the same time, various features have also been added to the .NET Framework till its current avatar (as of this writing) - .NET Framework 4.6.2. So much has happened with the .NET Framework till date that it probably runs a risk of eventually following the famous anti-pattern – the big ball of mud. It was about time that the .NET Framework became leaner, reliable and modular to make it ready for future needs.

.NET Core 1.0, released on 27 June 2016, is a step towards that direction.

> *In this article, we will explore .NET Core – a subset of the .NET Framework. The .NET Framework has been around for a while, and .NET Core is an attempt to make it more streamlined for specific purposes, structured enough to be lean and mean, and robust eventually. We will explore all these aspects in this article.*

.NET Core Architecture - Key Considerations

Let us look at how and why is the .NET Core envisioned. Here are some key considerations for the thought process behind .NET Core:

Fragmentation

.NET Framework ended up having many flavors to cater to the changes in the application delivery platforms (desktop, web, mobile apps, RIA - Silverlight), as well as cater to the Windows ecosystem changes (Windows App Store, Universal Windows Platform). It all started with the .NET Compact Framework actually. The fragmentation resulted in separate silos of .NET Framework implementations over a period of time. There is no problem per se with this approach. However, in today's world, when the overall technical architecture warrants to have apps supported on various devices, and of various kinds; it results into a nightmare for developers to code for various devices or app types (web, desktop or mobile). This was partially addressed earlier through portable class libraries (PCLs) and shared projects.

.NET core has now addressed this issue better, by having device specific implementations of runtimes.

The mscorblib conundrum

By design, quite a lot of features are bundled into mscorlib itself. This has advantages and disadvantages too. Given that now .NET Framework is envisioned to be lean and mean, and catering to various needs; a single installation per machine with an overload of various features does not become a scalable model going forward. So it would actually help to bundle only must-have features of .NET in order to make it lean and mean.

.NET Core will take care of this aspect more effectively now. .NET Core 1.0 (12 MB) is much smaller than the .NET Framework (200MB) and consists of smaller NuGet packages. The goal is to include components that your application really needs. For eg: with the .NET Framework, the Console class available in mscorlib was referenced by every .NET application, be it a Console, Web or a Windows application. However with .NET Core, if you want the Console functionality, you have to explicitly use the `System.Console` NuGet package. This keeps the framework lean and modular.

Framework Deployment

.NET Framework gets deployed as a single entity for every machine. This poses a problem when it comes to upgrading the Frameworks. While utmost care is taken by Microsoft to ensure backward compatibility without any regression, this approach causes an issue in terms of flexibility at design level (i.e. not being able to change behavior of interfaces etc). This issue is handled well now by componentizing .NET Core. Since there are fewer dependencies, the deployment includes only those components that your application really needs.

Embracing the OSS world – truly

Microsoft has embraced the open source world more effectively of late, and the efforts are very well seen through various open source technologies becoming available on Azure, as well as Integrated Development Environments like Visual Studio and Visual Studio Code. In context of the .NET Framework, while Mono was available as an open source version of .NET Framework, both Mono and .NET Framework were not actually the "same" frameworks when it came to the source-code. Sure, the functionalities were similar; but the code base was a ported version.

With .NET Core, this is issue is being addressed appropriately. Since .NET Core and ASP.NET are now

open source and platform-independent (supports Windows, Linux, and Mac systems); organizations that had earlier decided against running ASP.NET on their servers because it didn't support Linux, are now looking at ASP.NET Core, as it can now run on Linux.

Performance

.NET Core can be built to native code using .NET Native, not only on Windows, but also on Mac and Linux systems. Since C# code is compiled to native CPU instructions ahead-of-time (AoT) using the CoreRT runtime, it leads to performance improvements especially when the program is booting, as well as reduces the memory footprint for applications.

So these were some of the key considerations that are being taken into account to envision .NET Core as a way forward to the future .NET Framework.

So what exactly is .NET Core?

.NET Core simply put is a subset and a modular implementation of .NET Framework that can be used for delivering .NET based applications for specific needs viz. web apps, touch-based mobile apps and so on.

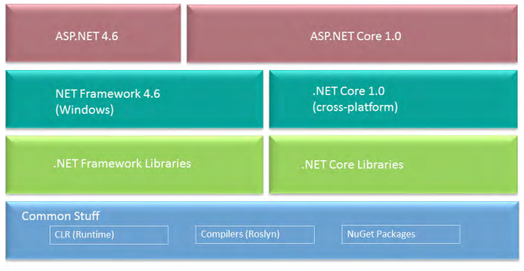


Figure 1: .NET Core in NET Framework Ecosystem

Here are some salient points of this framework (relates to v1.0):

1) .NET Core is a forked-out version of the .NET Framework codebase, and it focuses on web and mobility based services initially. This is supposed to get extended further in upcoming versions of .NET Core. The probable additions would be Xamarin based mobile applications, UWP (Universal Windows Platform), Windows applications etc.

2) It can service web app development through ASP.NET 1.0 Core, and will also support mobile/touch based development via .NET Native.

3) This forms a common baseline for running .NET across multiple platforms (hence the mention of Mac & Linux in Figure 1).

4) It banks on the CLR implementation for .NET Framework for consistency and reliability of various offerings of .NET Framework. It also uses appropriate compiler infrastructure to ensure that the coverage of languages supported is ensured.

5) NuGet packages are a new addition to the mix. This is a fundamental shift towards making .NET more modular through a package driven approach, and it also depicts the overall thinking process followed by Microsoft of late of making .NET a community driven, open-source based Framework in the future. Since .NET Core is based on small NuGet packages, changes can be implemented more easily.

As it is evident from various features, .NET Core is certainly a step towards Microsoft's journey to make .NET Framework a Standard subsequently.

.NET Core - what it is & what it is not?

Now that we have got an idea about the overall architecture of .NET, let us dive deeper to understand what .NET Core is, and what it is not. This is particularly important in order to understand how and why .NET Core has specific features, and also why it doesn't have some of the .NET Framework features.

So, .NET Core is:

1) Open Source: Going with the newly adopted philosophy at Microsoft, it is a no brainier that .NET Core is open source. It is available on GitHub under standard GitHub MIT license.

2) Cross Platform: .NET Core has been recently launched to run on Red Hat Enterprise (more details here). This gives a clear indication of how serious Microsoft is about the cross-platform commitment when it comes to .NET Core. Currently .NET Core runs on Windows, MacOS and of course Linux. A detailed list of supported OSes can be found here.

3) Flexible: The design goal of .NET Core ensures that you can include it in your app as embedded runtime, or it can be used as a usual machine-wide framework too (like .NET Framework)

4) Compatibility: It is fully compatible with .NET Framework, Mono and Xamarin.

5) NuGet Support: Realizing the issues with tight-bundling of .NET Framework, Microsoft had already started using packaged class libraries, and NuGet was used as the vehicle to drive this effort. Microsoft plans to use the same vehicle going forward for .NET Core too.

6) API Coverage: While there are many APIs that are common between .NET Core and .NET Framework, a lot has been removed as well, especially those APIs that are tightly bound to Windows like WinForms and WPF. The beauty of this framework however lies in being able to bundle only those APIs that your application would need (of course, to certain extent).

7) Command-line: This is something new and interesting. .NET Core comes with a lot of command-line tools such as "donet.exe" and so on for performing various tasks. A detailed list of all commands can be found here.

And .NET Core is not:

1) It is NOT another or newer version of .NET Framework. Let us just be absolutely clear about it.

2) It does not come pre-bundled with UI development for WPF. You will need to use .NET Framework for that. So nothing much changes really for desktop based development.

3) .NET Core is not designed with the focus of being smaller than the .NET Framework. Rather the aim is to create a framework that uses only those components that the application needs.

.NET Framework vs. .NET Core

Since .NET Core is essentially a different way of looking at overall Framework architecture of .NET, it obviously comes with some preconditions or changes.

1) Reflection support is going to be different. It is generally looked upon as a very expensive feature to manage when it comes to code compilation, and the output of compiler. So it may happen that Reflection as a feature either may not get supported in .NET Core, or will have a toned-down version of it included in future versions, or may eventually become an optional component.

2) App Domains are not supported in .NET Core. This has been done to ensure that it is a leaner implementation.

3) Remoting was more of a requirement to handle the architectures that had remote object invocation requirements. It also suffered from issues like performance and tight-coupling to an extent. Use of Remoting has reduced recently, and most of the requirements are now handled through loose coupling using WCF etc. So Remoting is more or less a forgotten technology, and hence has not made it to .NET Core.

4) Serialization of objects, contracts, XML etc. would be supported. However binary serialization wouldn't be a part of .NET Core.

5) Sandboxing is another feature that is not considered as a first class citizen of new .NET Core world at this point.

The .NET Portability Analyzer will come in handy to determine if an assembly is available or not, and recommend a workaround.

.NET Core Architecture

Now that we have a fair idea and background of what .NET Core is, let us spend some time on the architecture of .NET Core. Please note that since .NET Core is evolving, there could be some changes to this architecture in the future.

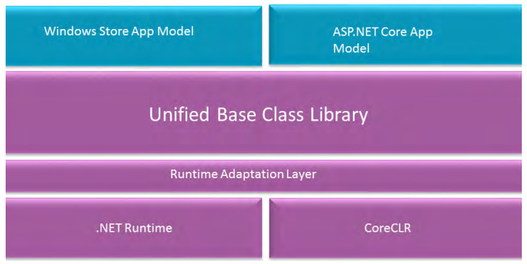


Image loosely based on the following source

There are some key aspects that you can decipher from the architecture. The runtimes shown in Figure 2 will be serving the following purposes:

1) CoreCLR is the runtime that is optimized for multi-platform and cloud base deployments. You will be using this when you are building applications on ASP.NET Core.

2) .NET Native (also referred as MRT – Managed Runtime – at times) is the CLR for Universal Apps and is optimized to run on any device and platform natively, which runs Windows. This will follow AOT (ahead of time) compilation instead of erstwhile JIT compilation, and will be targeted for specific platforms say Linux-64 bit and so on. This will ensure a better runtime performance.

Unified BCL (Base Class Library) consists of the basic/fundamental classes that are a part of .NET Core. This also is called as CoreFX. .NET Core follows the NuGet model for delivery of packages for BCL.

As per the design goals of .NET Core, BCL has various app models on top of it. Currently .NET Core has the Windows Store App model and ASP.NET Core App model enabling developers to build applications for targeting appropriate platforms.

A little bit more about ASP.NET Core

All along, we have been discussing about .NET Core. It would be unfair to not breifly mention about ASP.NET Core since both go hand-in-hand. .NET Core currently allows you to build Universal Store Apps and ASP.NET Core apps. ASP.NET Core is a complete rewrite of ASP.NET, and combines MVC and Web API into a unified platform running on the .NET Core.
Editorial Note: A good primer on ASP.NET Core can be found over here bitly.com/dnc-aspnet-core

.NET Core & App Development

All this is fine, but what happens to the current set of apps that you've already built with the .NET Framework? Do they have to work with .NET Core? Will they work on .NET Core? Well, let's find out.

ASP.NET Apps

Here are a few important points to remember while create ASP.NET apps:

1) .NET Core supports the ASP.NET MVC application architecture. However if your apps are Web Forms based, then probably you can give .NET Core a pass, since Web Forms is not a part of the .NET Core. Having said that, porting to .NET Core might be a good option if you plan to refactor your existing apps. Why? Probably the next point will explain this.

2) If you have built applications using ASP.NET MVC, then the sole reason for you to port them to .NET Core is the cross-platform compatibility that .NET Core brings to the table. If this specific aspect is not important to you, or is not a part of your overall product map, then you are just fine with your current .NET Framework based set up.

3) ASP.NET Core is decoupled from the web server. While it supports IIS and IIS Express, for self-hosting scenarios, it comes with a brand new high performance, cross-platform Web Server called *Kestrel*.

UWP Apps

If you have built Universal Windows Apps (UWP) on Windows 8 or Windows Phone 8.1, then they are already built on .NET Core. This also covers various form factors that Windows runs on viz. tablets, phones and so on.

Desktop Apps

.NET Core is still not ready for desktop apps yet. For the time being, we have to wait and watch.

Silverlight

Well if you have Silverlight apps, then probably you will eventually need to move them to HTML5 based apps, if you haven't already. A wise decision would be to look at .NET Core during this migration process, as one key design/architectural consideration in context of the app would be if it has to run cross platform.

Console Apps

You may have built some console apps on .NET Framework. They are the right candidates to be ported to .NET Core in order to support multiple OSes.

Should I take the plunge?

Oh yes, of course. With all the exciting features, ecosystem etc., you should certainly consider .NET Core. V1 of .NET Core is now available along with ASP.NET Core 1.0 and Entity Framework 1.0. You will need VS 2015 Update 3 and later, in order to build applications that use .NET Core 1.0.

Conclusion:

While the .NET Framework was designed to be cross-platform, practically it was always a Windows-only platform. .NET Core is a truly cross-platform framework. With the numerous versions of .NET Framework released so far, it is quite evident that Microsoft has been thinking through the overall design and architecture of .NET Framework from future-readiness perspective, and has been continuously improving the framework to stay up-to-date with different form-factors, ease of use, and rapid development of applications across platforms. While we are not there yet, .NET Core certainly seems to be an effort to make it the .NET Framework of Future.

The future indeed looks bright with .NET Core! Happy coding! ■

About the Author

rahul sahasrabuddhe

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

Damir Arh

Async and Await
Best Practices

The async and await keywords have been a part of C# language since version 5.0, which was released in autumn 2013 as part of Visual Studio 2013. Although in essence they make asynchronous programming simpler, one can still use them incorrectly. This article describes some of these common mistakes, and gives advice on how to avoid them.

The Basics of C# Async and Await

As users, we prefer applications which respond quickly to our interactions, and do not "freeze" when loading or processing data. While it might still be acceptable for legacy desktop line of business applications to stop responding occasionally, we are less patient with mobile applications that keep us waiting. Even operating systems are becoming more responsive, and give you the option to terminate such misbehaving processes.

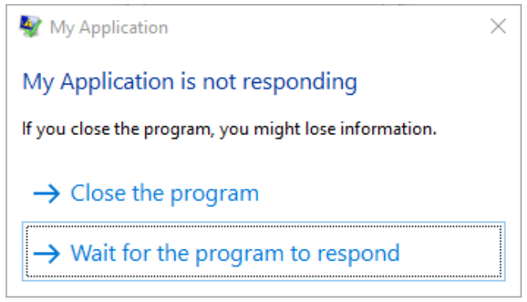


Figure 1: Windows warns about applications that stopped responding

If you have been a developer for a while, it is very likely that you faced scenarios where your application became unresponsive. One such example is a data retrieval operation that usually completed in a few hundred milliseconds at the most, suddenly took several seconds because of server or network connectivity issues. Since the call was synchronous, the application did not respond to any user interaction during that time.

To understand why this happens, we must take a closer look at how the operating system communicates with applications. Whenever the OS needs to notify the application of something, be it the user clicking a button or wanting to close the application; it sends a message with all the information describing the action to the application. These messages are stored in a queue, waiting for the application to process them and react accordingly.

Each application with a graphical user interface (GUI) has a main message loop which continuously checks the contents of this queue. If there are any unprocessed messages in the queue, it takes out the first one and processes it. In a higher-level language such as C #, this usually results in invoking a corresponding event handler. The code in the event handler executes synchronously. Until it completes, none of the other messages in the queue are processed. If it takes too long, the application will appear to stop responding to user interaction.

Asynchronous programming using `async` and `await` keywords provides a simple way to avoid this problem with minimal code changes. For example, the following event handler synchronously downloads an HTTP resource:

```
private void OnRequestDownload(object
sender, RoutedEventArgs e)
{
    var request = HttpWebRequest.
Create(_requestedUri);
    var response = request.
GetResponse();
    // process the response
}
```

Here is an asynchronous version of the same code:

```
private async void
OnRequestDownload(object sender,
RoutedEventArgs e)
{
    var request = HttpWebRequest.
Create(_requestedUri);
    var response = await request.
GetResponseAsync();
    // process the response
}
```

Only three changes were required:

- The method signature changed from `void` to `async void`, indicating that the method is asynchronous, allowing us to use the `await` keyword in its body.

- Instead of calling the synchronous `GetResponse` method, we are calling the asynchronous `GetResponseAsync` method. By convention, asynchronous method names usually have the `Async` postfix.

- We added the await keyword before the asynchronous method call.

This changes the behavior of the event handler. Only a part of the method up to the `GetResponseAsync` call, executes synchronously. At that point, the execution of the event handler pauses and the application returns to processing the messages from the queue.

Meanwhile, the download operation continues in the background. Once it completes, it posts a new message to the queue. When the message loop processes it, the execution of the event handler method resumes from the `GetResponseAsync`

call. First, its result of type `Task<WebResponse>` is unwrapped to `WebResponse` and assigned to the `response` variable. Then, the rest of the method executes as expected. The compiler generates all the necessary plumbing code for this to work.

Although Web applications are built and run using a single UI thread, they can still benefit from asynchronous programming. A dedicated thread processes each incoming request. While this thread is busy with one request, it cannot start processing another one. Since there is a limited number of threads available in the thread pool, this limits the number of requests that can be processed in parallel. Any thread waiting for an I/O operation to complete, is therefore a wasted resource. If the I/O operation is performed asynchronously instead, the thread is not required any more until the operation completes, and is released back to the thread pool, making it available to process other requests. Although this might slightly increase the latency of a single request, it will improve the overall throughput of the application.

The use of `async` and `await` keywords is not limited to asynchronous programming though. With Task Parallel Library (TPL) you can offload CPU intensive operations onto a separate thread by calling `Task.Run`. You can await the returned task the same way as you would with an asynchronous method to prevent blocking the UI thread. Unlike real asynchronous operations, the offloaded work still consumes a thread; therefore, this strategy is not as useful in web applications where there is no special thread to keep available.

Async Await Common Pitfalls

The `async` and `await` keywords provide a great benefit to C# developers by making asynchronous programming easier. Most of the times, one can use them without having to understand the inner workings in detail. At least, as long as the compiler is a good enough validator, the code will behave as intended. However, there are cases when incorrectly written asynchronous code will compile successfully, but still introduce subtle bugs that can be hard to troubleshoot and fix.

Let us look at some of the most common pitfall examples.

Avoid Using Async Void

The signature of our asynchronous method in the code example we just saw was `async void`.

While this is appropriate for an event handler and the only way to write one, you should avoid this signature in all other cases. Instead, you should use `async Task` or `async Task<T>` whenever possible, where T is the return type of your method.

As explained in the previous example, we need to call all asynchronous methods using the await keyword, e.g.:

```
DoSomeStuff(); // synchronous method
await DoSomeLengthyStuffAsync(); //
long-running asynchronous method
DoSomeMoreStuff(); // another
synchronous method
```

This allows the compiler to split the calling method at the point of the `await` keyword. The first part ends with the asynchronous method call; the second part starts with using its result if any, and continues from there on.

In order to use the `await` keyword on a method; its return type must be `Task`. This allows the compiler to trigger the continuation of our method, once the task completes. In other words, this will work as long as the asynchronous method's signature is `async Task`. Had the signature been `async void` instead, we would have to call it without the `await` keyword:

```
DoSomeStuff(); // synchronous method
DoSomeLengthyStuffAsync(); // long-
running asynchronous method
DoSomeMoreStuff(); // another
synchronous method
```


The compiler would not complain though. Depending on the side effects of `DoSomeLengthyStuffAsync`, the code might even work correctly. However, there is one important difference between the two examples. In the first one, `DoSomeMoreStuff` will only be invoked after `DoSomeLengthyStuffAsync` completes. In the second one, `DoSomeMoreStuff` will be invoked immediately after `DoSomeLengthyStuffAsync` starts. Since in the latter case `DoSomeLengthyStuffAsync` and `DoSomeMoreStuff` run in parallel, race conditions might occur. If `DoSomeMoreStuff` depends on any of `DoSomeLengthyStuffAsync`'s side effects, these might or might not yet be available when `DoSomeMoreStuff` wants to use them. Such a bug can be difficult to fix, because it cannot be reproduced reliably. It can also occur only in production environment, where I/O operations are usually slower than in development environment.

To avoid such issues altogether, always use `async Task` as the signature for methods you intend to call from your code. Restrict the usage of `async void` signature to event handlers, which are not allowed to return anything, and make sure you never call them yourself. If you need to reuse the code in an event handler, refactor it into a separate method returning `Task`, and call that new method from both the event handler and your method, using `await`.

Beware of Deadlocks

In a way, asynchronous methods behave contagiously. To call an asynchronous method with `await`, you must make the calling method asynchronous as well, even if it was not async before. Now, all methods calling this newly asynchronous method must also become asynchronous. This pattern repeats itself up the call stack until it finally reaches the entry points, e.g. event handlers.

When one of the methods on this path to the entry points cannot be asynchronous, this poses a problem. For example, constructors. They cannot be asynchronous, therefore you cannot use `await` in their body. As discussed in the previous section, you could break the asynchronous requirement early by giving a method `async void` signature, but this prevents you from waiting for its execution to end, which makes it a bad idea in most cases.

Alternatively, you could try synchronously waiting for the asynchronous method to complete, by calling `Wait` on the returned Task, or reading its `Result` property. Of course, this synchronous code will temporarily stop your application from processing the message queue, which we wanted to avoid in the first place. Even worse, in some cases you could cause a deadlock in your application with some very innocent looking code:

```
private async void MyEventHandler(object
sender, RoutedEventArgs e)
{
  var instance = new
  InnocentLookingClass();
  // further code
}
```

Any synchronously called asynchronous code in `InnocentLookingClass` constructor is enough to cause a deadlock:

```
public class InnocentLookingClass()
{
  public InnocentLookingClass()
  {
    DoSomeLengthyStuffAsync().Wait();
    // do some more stuff
  }

  private async Task
  DoSomeLengthyStuffAsync()
  {
    await SomeOtherLengthyStuffAsync();
  }

  // other class members
}
```

Let us dissect what is happening in this code.

`MyEventHandler` synchronously calls `InnocentLookingClass` constructor, which invokes `DoSomeLengthyStuffAsync`, which in turn asynchronously invokes `SomeOtherLengthyStuffAsync`. The execution of the latter method starts; at the same time the main thread blocks at Wait until

`DoSomeLengthyStuffAsync` completes without giving control back to the main message loop. Eventually `SomeOtherLengthyStuffAsync` completes and posts a message to the message queue implying that the execution of `DoSomeLengthyStuffAsync` can continue. Unfortunately, the main thread is waiting for that method to complete instead of processing the messages, and will therefore never trigger it to continue, hence waiting indefinitely.

As you can see, synchronously invoking asynchronous methods can quickly have undesired consequences. Avoid it at all costs; unless you are sure what you are doing, i.e. you are not blocking the main message loop.

Allow Continuation on a Different Thread

The deadlock in the above example would not happen if `DoSomeLengthyStuffAsync` did not require to continue on the main thread where it was running before the asynchronous call. In this case, it would not matter that this thread was busy waiting for it to complete, and the execution could continue on another thread. Once completed, the constructor execution could continue as well.

As it turns out, there is a way to achieve this when awaiting asynchronous calls – by invoking `ConfigureAwait(false)` on the returned `Task` before awaiting it:

```
await SomeOtherLengthyStuffAsync().
ConfigureAwait(false);
```

This modification would avoid the deadlock, although the problem of the synchronous call in the constructor, blocking the message loop until it completes, would remain.

While allowing continuation on a different thread in the above example might not be the best approach, there are scenarios in which it makes perfect sense. Switching the execution context back to the originating thread affects performance, and as long as you are sure that none of the code after it resumes needs to run on that thread, disabling the context switch will make your code run faster.

You might wonder which code requires the context to be restored. This depends on the type of the application:

- For user interface based applications (Windows Forms, WPF and UWP), this is required for any code that interacts with user interface components.
- For web applications (ASP.NET), this is required for any code accessing the request context or authentication information.

When you are unsure, you can use the following rule of thumb, which works fine in most cases:

- Code in reusable class libraries can safely disable context restoration.

- Application code should keep the default continuation on the originating thread – just to be on the safer side.

Planned Features for C# 7.0

The language designers succeeded in making `async` and `await` very useful with its first release in 2013. Nevertheless, they are constantly paying attention to developer feedback, and try to improve the experience wherever and whenever it makes sense.

For example, in C# 6.0 (released with Visual Studio 2015) they added support for using await inside `catch` and `finally` blocks. This made it easier and less error prone to use asynchronous methods in error handling code.

According to plans (which might however still change), C# 7 will have two new features related to asynchronous programming.

Note: For those new to C# 7, make sure to check out C# 7 – Expected Features

Support for Async Main

In the section about deadlocks, I explained how making a method asynchronous has a tendency of propagating all the way to their entry points. This works out fine for event driven frameworks (such as Windows Forms and WPF) because event handlers can safely use `async void` signature, and for ASP.NET MVC applications, which support asynchronous action methods.

If you want to use asynchronous methods from a simple console application, you are on your own. Main method as the entry point for console applications must not be asynchronous. Hence to call asynchronous methods in a console application, you need to create your own top-level asynchronous wrapper method and call it synchronously from Main:

```
static void Main()
{
  MainAsync().Wait();
}
```

Since there is no message loop to block in a console application, this code is safe to use without the danger of causing a deadlock.

Language designers are considering the idea of adding support for asynchronous entry points for console applications, directly into the compiler. This would make any of the following method signatures valid entry points for console applications in C# 7.0:

```
// current valid entry point signatures
void Main()
int Main()
void Main(string[])
int Main(string[])
// proposed additional valid entry point
signatures in C# 7.0
async Task Main()
async Task<int> Main()
async Task Main(string[])
async Task<int> Main(string[])
```

While this feature might not enable anything that is not already possible, it will reduce the amount of boilerplate code and make it easier for beginners to call asynchronous methods correctly from console applications.

Performance Improvements

I have already discussed the impact of context switching on asynchronous programming with `async` and `await`. Another important aspect are memory allocations.

Each asynchronous method allocates up to three objects on the heap:

- the state machine with method's local variables,
- the delegate to be called on continuation, and
- the returned `Task`.

Since additional object allocations boils down to more work for the garbage collector, the current implementation is already highly optimized. The first two allocations only happen when they are required, i.e. when another asynchronous method is actually awaited. E.g. this scenario would only occur to the following method when called with `true`:

```
private async Task DoSomeWorkAsync(bool
doRealWork)
{
  if (doRealWork)
  {
    await DoSomeRealWorkAsync();

  }
}
```

This is a contrived example, but even real-world methods often include edge cases with different execution paths, which might skip all asynchronous calls in them.

The allocation of the returned task is also already somewhat optimized. Common Task objects (for values `0`, `1`, `true`, `false`, etc.) are cached to avoid allocating a new one whenever one of these commonly used values are returned.

C# 7.0 promises to bring this optimization a step further. Asynchronous methods returning value types will be able to return `ValueTask<T>` instead of `Task<T>.` As the name implies, unlike `Task<T>`, `ValueTask<T>` is itself a `struct`, i.e. a value type that will be allocated on the stack instead of on the heap. This will avoid any heap

allocations whatsoever for asynchronous methods returning value types, when they make no further asynchronous calls.

Apart from the benefit for garbage collection, initial tests by the team also show almost 50% less time overhead for asynchronous method invocations, as stated in the feature proposal on GitHub. When used in tight loops, all of this can add up to significant performance improvements.

Conclusion:

Even though asynchronous programming with `async` and `await` seems simple enough once you get used to it, there are still pitfalls to be aware of. The most common one is improper use of `async void`, which you can easily overlook and the compiler will not warn you about it either. This can introduce subtle and hard to reproduce bugs in your code that will cost you a lot of time to fix. If being aware of that remains your only takeaway from this article, you have already benefited. Of course, learning about the other topics discussed in this article, will make you an even better developer ■

About the Author

damir arh

Damir Arh has many years of experience with Microsoft development tools; both in complex enterprise software projects and modern cross-platform mobile applications. In his drive towards better development processes, he is a proponent of test driven development, continuous integration and continuous deployment. He shares his knowledge by speaking at local user groups and conferences, blogging, and answering questions on Stack Overflow. He is an awarded Microsoft MVP for .NET since 2012.

Microsoft® MVP Most Valuable Professional

Daniel Garcia

A Vanity Look at ASP.NET Core

ASP.NET Core is finally here! On June 27th Microsoft released v1.0 of .Net Core, ASP.NET Core and Entity Framework Core, including a preview of command line tools, as well as Visual Studio and Visual Studio Code extensions to create .NET Core applications.

We now have in our hands a cross-platform, open source and modular .NET Platform that can be used to build modern web applications, libraries and services.

Throughout this article, we will build **a simple web application to demonstrate some of the changes in ASP.NET Core, like its new request pipeline**. This will allow us to explore some of the new features and changes in action, hopefully making them easier to understand. Don't be concerned, your past experience working with the likes of MVC or Web API is still quite relevant and helpful.

The main purpose of the web application will be allowing public profiles of registered users to be associated with vanity urls. That is, if I select /daniel.jimenez as my vanity url, then visitors will be able to find my profile navigating to my-awesome-site.com/daniel.jimenez.

***Note**: For those who are new to the concept of a vanity URL, it is a customized short URL, created to brand a website, person, item and can be used in place of traditional longer URLs.*

I hope you find this article as interesting to read, as it was for me to write!

ASP.NET Core (previously known as ASP.NET 5) is the latest .NET stack to create modern web applications. It is a complete rewrite of ASP.NET and is open source, encourages modular programming, is cloud enabled, and is available across non-Microsoft platforms like Linux and Mac. In this stack, you have ASP.NET MVC, WEB API and Web Pages which have been unified and merged into one single unified framework called as MVC 6. ASP.NET Web Forms is not a part of ASP.NET Core 1.0, although the WebForms team is actively maintaining the web forms framework.

Laying the foundations

Creating a new web application

We could start completely from scratch when creating a new ASP.NET Core application, but for the purposes of this article I will start from one of the templates installed in Visual Studio 2015 as part of the tooling.

Note: If you are using Visual Studio 2015 / Visual Studio Community edition, get VS2015 Update 3 first and then install the .NET Core Tools for Visual Studio.

I have chosen the *ASP.NET Core Web Application* template including *Individual User Accounts* as the authentication method.

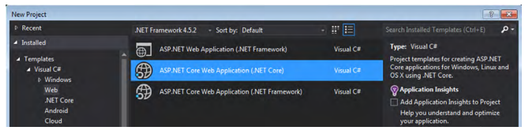


Figure 1. New project type

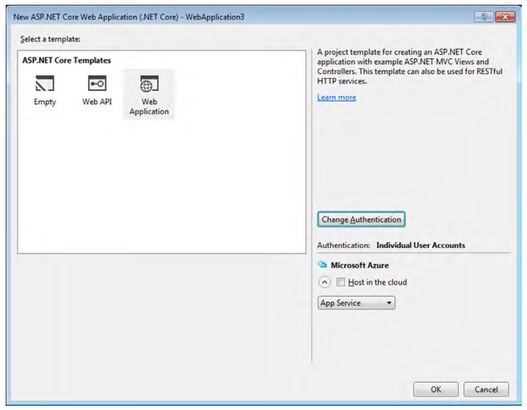


Figure 2. Starting from a Web Application with Individual User Accounts

This will give us a good starting point for our website. It is also worth mentioning that the authentication will be setup using the new ASP.NET Core Identity framework, including an Entity Framework Core context for storing user accounts.

If you are not using Visual Studio, you should be able to use the yeoman aspnet generators that are part of the OmniSharp project. Their templates are based on the ones included in Visual Studio, so its *Web Application* template provides a similar starting point.

Registering with a Vanity Url

Initialize the database

Once the new application has been created you should be able to launch it and navigate to **/Account/Register**, where you will see the initial page for registering new accounts.

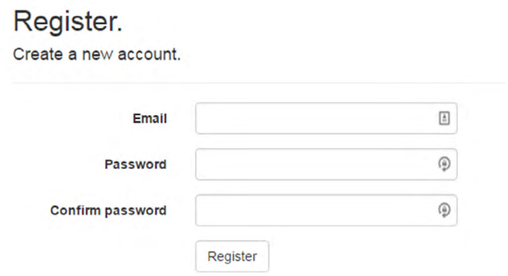


Figure 3. Default page for registering new accounts

If you go ahead and try to register, you will find out that your password needs at least one non alphanumeric character, one digit and one upper case letter. You can either match those requirements or change the password options when adding the Identity services in the **ConfigureServices** method of the **Startup** class.

Just for the sake of learning, let's do the latter, and take a first look at the Startup class, where you add and configure the independent modules that our new web application is made of. In this particular case, let's change the default Identity configuration added by the project template:

```
services.AddIdentity<ApplicationUser,
IdentityRole>(opts =>
{
  opts.Password.RequireNonAlphanumeric =
  false;
  opts.Password.RequireUppercase =
```

```
  false;
  opts.Password.RequireDigit = false;
})
```

Try again and this time you will see a rather helpful error page that basically reminds you to apply the migrations that initialize the identity schema:

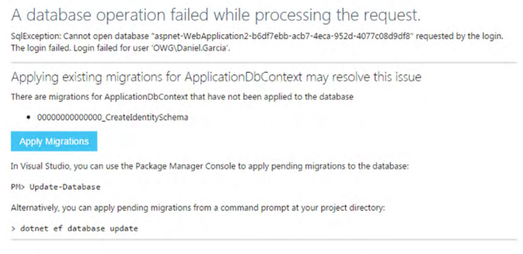


Figure 4.Error prior to applying migrations

Let's stop the application (or the command will fail) and run the suggested command in a console from the project root folder, the one containing the ***.xproj** file:

```
>dotnet ef database update
```

This will initialize a new database in your SQL Server Local DB instance, including the new Identity schema. If you try to register again, it should succeed and you will have created the first user account.

In Visual Studio, you can quickly open the SQL Server Object Explorer, open the localdb instance, locate the database for your web application, and view the data in the AspNetUsers table:

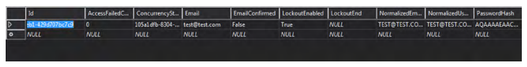


Figure 5. Registered user saved to localdb

Add a VanityUrl column to the schema

So far, so good. The template gave us a good starting point and we have a web application where users can register by creating an account in the database. The next step will be updating the application so users can pick a vanity url when registering.

First we are going to add a new VanityUrl field to the **ApplicationUser** class, which is the simplest way of adding additional properties to profiles. We will add the property as per the requirement, that is a max length of 256 characters (If you want, you can also go ahead and add additional fields like first name, last name, DoB, etc.):

```
public class ApplicationUser :
IdentityUser
{
  [Required, MaxLength(256)]
  public string VanityUrl { get; set; }
}
```

Now we need to add a new EF migration (entity framework), so these schema changes get reflected in the database. In the process, we will also add a unique index over the new VanityUrl column. Since we will need to find users given their vanity url, we better speed up those queries!

To add the migration, run the following command from the project root:

```
>dotnet ef migrations add
VanityUrlColumn
```

This will auto-generate a migration class, but the database won't be updated until you run the update command. Before that, make sure to update the migration with the unique index over the vanity url field:

```
public partial class VanityUrlColumn :
Migration
{
 protected override void
 Up(MigrationBuilder migrationBuilder)
 {
  migrationBuilder.AddColumn<string>(
   name: "VanityUrl",
   table: "AspNetUsers",
   maxLength: 256,
   nullable: false,
   defaultValue: "");
  migrationBuilder.CreateIndex(
   "IX_AspNetUsers_VanityUrl",
   "AspNetUsers", "VanityUrl",
   unique: true);
 }

 protected override void
 Down(MigrationBuilder migrationBuilder)
 {
  migrationBuilder.DropColumn(
   name: "VanityUrl",
   table: "AspNetUsers");
  migrationBuilder.DropIndex("IX_
  AspNetUsers_VanityUrl");
 }
}
```

Finally go ahead and update the database:

```
>dotnet ef database update
```

Update the Register page

Right now our Register page is broken and we are not able to create new user accounts. This makes sense since we added the VanityUrl column as required, but we haven't updated the register page to capture the new field. We will fix this right now.

Start by adding a new property to the existing **RegisterViewModel** class. As you might expect, we will add some attributes to make it a required field, allow 3 to 256 characters, and allow only lower case letters, numbers, dashes and dots:

```
[Required]
[StringLength(256,
  ErrorMessage = "The {0} must be at
  least {2} and at max {1} characters
  long.",
  MinimumLength = 3)]
[RegularExpression(@"[a-z0-9\.\-]+",
  ErrorMessage = "Use only lower case
letters, numbers, dashes and dots")]
[Display(Name = "Vanity Url")]
public string VanityUrl { get; set; }
```

Now update the existing view **Account\Register.cshtml**, adding the VanityUrl field to the form. If you have used MVC before, you might be surprised by the lack of Html helpers. They are still available but, ASP.NET Core has added **tag helpers** to its tool belt.

With tag helpers, you can write server side rendering code targeting specific html elements (either standard html tags or your own custom tags) that will participate in creating and rendering the final HTML from a razor file. This style of writing your server side views allows for more robust and readable code in your views, seamlessly integrating your server side rendering helpers within the html code.

The razor code for the VanityUrl field will be quite similar to that of the existing fields, using the tag helpers for rendering the label, input and validation message. We will add a bit of flashiness using a bootstrap input group displaying our website's host, so users can see what their full vanity url would look like:

```
<div class="form-group">
 <label asp-for="VanityUrl" class="col-
 md-2 control-label"></label>
 <div class="col-md-10">
  <div class="input-group">
   <span class="input-group-addon">@
   ViewContext.HttpContext.Request.
   Host/</span>
   <input asp-for="VanityUrl"
   class="form-control"/>
  </div>
  <span asp-validation-for="VanityUrl"
  class="text-danger"></span>
 </div>
</div>
```

Finally, update the **Register** action in the **AccountController**, so the VanityUrl is mapped from the **RegisterViewModel** to the **ApplicationUser.**

```
var user = new ApplicationUser {
 UserName = model.Email,
 Email = model.Email,
 VanityUrl = model.VanityUrl
};
```

Users are now able to provide a vanity url while registering, and we will keep that vanity url together with the rest of the user data:

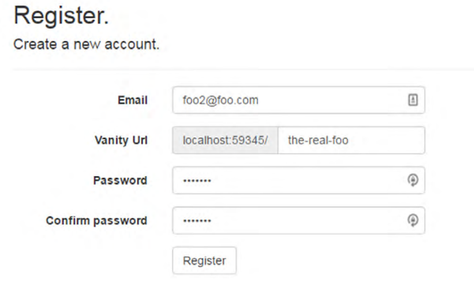


Figure 6. Updated register page

If you added additional profile fields (like first name, last name or DoB) to the `ApplicationUser` class, follow the same steps with those properties in order to capture and save them to the database.

Adding a public profile page

Right now users can create an account and enter a vanity url like **/the-real-foo**. The final objective will be associating those urls with a public profile page, but we will start by adding that page. It will initially be accessible only through the standard routing **/controller/action/id?**, leaving the handling of the vanity urls for the next section.

Create a new **ProfileController** with a single **Details** action that receives an id string parameter, which should match the id in the AspNetUsers table. This means using urls like *mysite.com/profile/details/b54fb19b-aaf5-4161-9680-7b825fe4f45a*, which is rather far from ideal. Our vanity urls as in *mysite.com/the-real-foo* will provide a much better user experience.

Next, create the **Views\Profile\Details.cshtml** view and return it from the controller action so you can test if the page is accessible:

```
public IActionResult Details(string id)

{
  return View();
}
```

Since we don't want to expose the **ApplicationUser** directly in that view (that would expose ids, password hashes etc.), create a new view model named **Profile**. Add any public properties from ApplicationUser that you want exposed, like the name or DoB. If you didn't add any extra properties let's just add the UserName and the VanityuUrl so we have something to show in the page.

You will need to update the **Profile\Details.cshtml** view so it declares the new Profile class as its model and renders its properties. For the sake of brevity I will skip this, you should be able to manually write your own, or use the Visual Studio wizard for adding a new view, selecting the Details template and our new Profile class as the view model. Please check the source code in GitHub if you find any problems. A more interesting change is required in the **ProfileController**, where we need to retrieve an **ApplicationUser** from the database given its id, and then map it to the new **Profile** class.

Using dependency injection in the Profile Controller

In order to retrieve an `ApplicationUser` from the database, the Identity framework already provides a class that can be used for that purpose, the **UserManager<ApplicationUser>**, which contains a **FindByIdAsync** method. But how do we access that class from our **ProfileController**? Here is where dependency injection comes.

Dependency Injection has been built into ASP.NET Core, and it is used by components like the Identity framework to register and resolve their dependencies. Of course, you can also register and resolve your own components. Right now, let's use constructor injection to receive an instance of the UserManager class in our controller constructor:

```
private readonly
UserManager<ApplicationUser> _
userManager;
public ProfileController
(UserManager<ApplicationUser>
userManager)
{
 _userManager = userManager;
}
```

If you set a breakpoint, you will see an instance is being provided. This is because in your **Startup** class, the line `services.AddIdentity()` has registered that class within the dependency injection container. When an instance of your controller needs to be created, the container realizes a `UserManager` is needed, providing an instance of the type that was previously registered. (You would get an exception if the required type is not registered)

Now update the action so it finds the user, creates a `Profile` instance and pass it to the view:

```
public async Task<IActionResult>
Details(string id)
{
 var user = await _userManager.
 FindByIdAsync(id);
 return View(new Profile
 {
   Email = user.Email,
   Name = user.UserName,
   VanityUrl = user.VanityUrl
 });
}
```

This completes the public profile page, although accessible only with the default routing. We will make sure that page can also be accessed using the vanity urls in the next section!

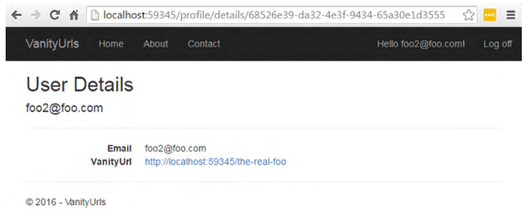


Figure 7. Public profile page with default routing

Handling the vanity urls

Middleware to the rescue!

The new request pipeline

The request pipeline in ASP.NET Core is one of the areas with the biggest number of changes. Gone is the request pipeline based on events, and gone are the HttpHandlers and HttpModules of old, that closely followed IIS features.

The new pipeline is leaner, composable and completely independent of the hosting solution. It is based around the concept of middleware, in which a pipeline can be composed of independent modules that receive the request, run its own logic then call the next module.

If you have worked with Express in Nodejs, this should sound very familiar!

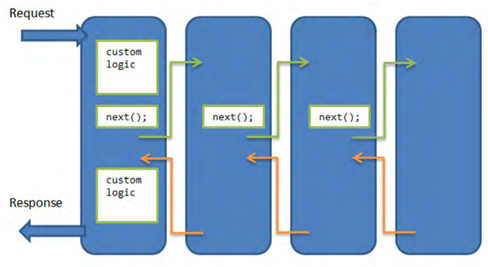


Figure 8. New request pipeline based on middleware

After receiving the response from the next middleware, you also have a chance to run custom logic, potentially updating/inspecting the response. Calling the next module is optional, so some of these modules (like authentication) might decide to end the request earlier than usual.

The ASP.NET Core framework provides a number of built-in middleware like Routing, Authentication or CORS (In case you are wondering, MVC is dependent on the Routing middleware, wired as a route handler). You can also create your own middleware classes or even add them inline as lambda functions.

The place where the middleware components are plugged in to form the request pipeline is the Configure method of the Startup class. As you can imagine, the order in which these are added to the pipeline is critical! The Startup class of this project contains by default:

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

Adding custom middleware for the vanity urls

Ok, so we can add our own middleware to the pipeline. How will that help us with the vanity urls?

The idea is to create a new piece of middleware that inspects the url before the routing middleware. If the url has a single segment, it could be one of our vanity urls, so we will query the database for a user account with that vanity url. In case we have a match, we are going to update the path in the *Request* object so it matches the standard route for the public profile page. This process will happen server side, without client redirections involved.

You might be wondering why are we using a middleware for this, and not the routing features? In ASP.NET Core, middleware components are the proper place to inspect and modify the request and/or response objects. This allows writing very simple and testable components following the single responsibility principle. These components can then be composed in several different ways in order to build the request processing pipeline. In our case:

- The new middleware component resolves vanity urls into a standard *controller/action/id* url.

- The routing middleware continues performing url pattern matching to find the right controller and action.

In summary, when we process a url like *mysite.com/the-real-foo*, our middleware component will find the foo user in the database, get its **ApplicationUser** object which includes its id, and then update the url in the request object to be *mysite.com/profile/details/b54fb19b-aaf5-4161-9680-7b825fe4f45a*. After that we will call `next();` to execute the next middleware in the pipeline, which means the routing middleware will be able to send the request to our **ProfileController**!

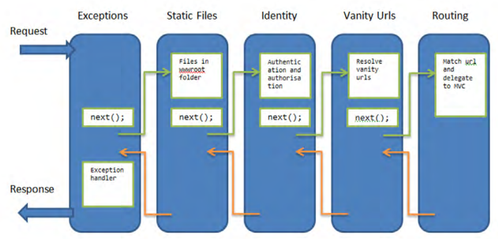


Figure 9. Request pipeline including the VanityUrl middleware

Let's go ahead and create a new class **VanityUrlsMiddleware** inside a new Middleware folder (notice how you can use dependency injection again to receive the UserManager):

```
public class VanityUrlsMiddleware
{
 private readonly RequestDelegate _next;
 private readonly
 UserManager<ApplicationUser> _
 userManager;
 public VanityUrlsMiddleware(
  RequestDelegate next,
  UserManager<ApplicationUser>
  userManager)
  {
    _next = next;
    _userManager = userManager;

  }
  public async Task Invoke(HttpContext
  context)
  {
   await HandleVanityUrl(context);
   //Let the next middleware (MVC
   routing) handle the request
   //In case the path was updated,
   //the MVC routing will see the
   updated path
   await _next.Invoke(context);
  }
  private async Task
  HandleVanityUrl(HttpContext context)
  {
   //TODO
  }
```

Now add it to the pipeline right before adding MVC in the **Configure** method of the Startup class:

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

If you run the application now, you should be able to set a breakpoint inside your middleware class, and it will be hit on every request.

Let's finish the middleware by adding the following logic to the **HandleVanityUrl** method:

1. Make sure the request path has a single segment and matches the regex for vanity urls. Remember we added a regex on the `RegisterViewModel` to allow only lower case letters, numbers, dashes and dots?

We can use the same regex here, extracting it as a constant.

2. Query the database, trying to find a user with that vanity url

3. Replace the *Path* property of the *Request* object with the pattern *profile/details/{id}*

This code will look like the following:

```
private async Task
HandleVanityUrl(HttpContext context)
{
 //get path from request
 var path = context.Request.Path.
 ToUriComponent();
 if (path[0] == '/')
 {
  path = path.Substring(1);
 }
 //Check if it matches the VanityUrl
 regex
 //(single segment, only lower case
 letters, dots and dashes)
 //Check accompanying sample project for
 more details
 if (!IsVanityUrl(path))
 {
  return;
 }

 //Check if a user with this vanity url
 can be found
 var user = await _userManager.Users.
 SingleOrDefaultAsync(u =>
 u.VanityUrl.Equals(path,
 StringComparison.
 CurrentCultureIgnoreCase));
 if (user == null)
 {
  return;
 }

 //If we got this far, the url matches a
 vanity url,
 //which can be resolved to the profile
 details page.
 context.Request.Path = String.Format(
  "/profile/details/{0}", user.Id);
}
```

That's it, now you have vanity urls working in your application! If you inspect the network tab of your browser, you will see there were no redirects, it all happened seamlessly on the server:

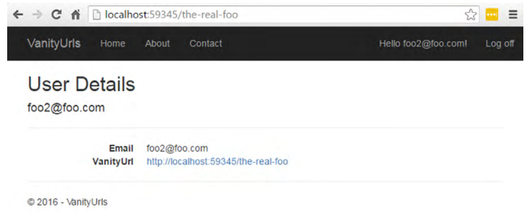


Figure 10. The same profile page accessed with the vanity url

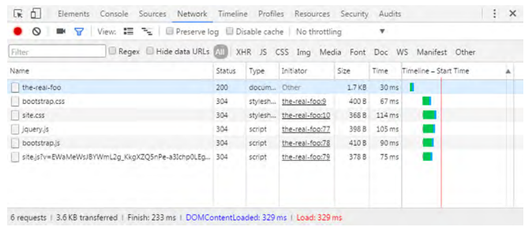


Figure 11. Requests when accessing a vanity url

Validate user selected urls

You might remember that when we added the VanityUrl column to the database, we created a Unique Index. This means an exception is raised if you try to register a new user with a vanity url already in use.

This isn't the world's best user experience, although it might be enough for the purposes of the article. However the **Remote** attribute is still available which means we can quickly improve our application!

Note: If you are not familiar with previous versions of ASP MVC, the Remote attribute is one of the built in validation attributes that adds client and/or server side validations. This particular attribute adds a client side validation which will call the specified controller action.

We just need to add a new controller action that

returns a JSON indicating whether the value matches an existing vanity url or not. I decided to add this action within the *ProfileController* so the validation logic stays close to its use case, but the `AccountController` will be an equally valid option:

```
public async Task<JsonResult>
ValidateVanityUrl(string vanityUrl)
{
  var user = await _userManager.Users.
  SingleOrDefaultAsync(
    u => u.VanityUrl == vanityUrl);
    return Json(user == null);
}
```

Then add the **Remote** attribute to the property in the **RegisterViewModel**:

```
..
[Remote("ValidateVanityUrl", "Profile",
ErrorMessage = "This vanity url is
already in use")]
…
public string VanityUrl { get; set; }
```

Wrapping up

Improvements

Since we have already covered a lot of ground, I have left out a few improvements that can be easily applied to this code. Feel free to get the source from GitHub and play with it:

- Every time a vanity url is resolved, the `ApplicationUser` is retrieved twice from the database. This can be improved, for example by simply saving the `ApplicationUser` to the `HttpContext.Items`, and updating the `ProfileController` to first check that collection before hitting the database again. Furthermore, this can be moved to a `ProfileService` injected into the controller.

- The middleware has the resolved url pattern *profile/details/{0}* hardcoded internally. This is a good excuse to explore the Configuration and Options libraries built into ASP.NET Core!

- The logic to find an `ApplicationUser` given its vanity url is duplicated in two different places. This is a good candidate to be extracted into a `UserService`, injected in those places.

Conclusion

We have just seen the tip of the iceberg in terms of what ASP.NET Core has to offer. It contains fundamental changes like its new request pipeline or built-in dependency injection, and improvements not that disruptive as the tag helpers.

However ASP.NET Core should be very familiar and easy to get started for developers used to the *old* ASP.Net. You should be able to start working with a leaner, cross-platform, composable framework without your current knowledge becoming completely obsolete. In fact, you should be able to transfer most of that knowledge.

For further reading, I would recommend checking the new official documentation site and the framework GitHub page ■

Download the entire source code from GitHub at bit.ly/dncm26-aspcorevanity

About the Author

daniel garcia

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

Christian Del Bianco

BROADCAST REAL-TIME NOTIFICATIONS

using SignalR, Knockout JS and SqlTableDependency

ASP.NET SignalR is an open source library that adds real-time notifications to our application by establishing a two-way communication channel between server and client, in order to send asynchronous messages in both directions.

HTML5 specifications about WebSocket have been introduced specifically to meet this need. Although natively supported by ASP.NET 4.5, as well as in the latest versions of popular browsers, it does requires the presence of the IIS 8 web server. This poses a challenge especially when we are building a site intended for diverse audiences, since we cannot be absolutely sure that all our visitors will use browsers that are compliant with this technology.

However, there are other techniques to establish a full-duplex communication channel and luckily ASP.NET SignalR can help us with that. During the initial stages of the communication and based on the characteristics of the server and the client, SignalR automatically makes a decision about the most suitable connection strategy to use, choosing between:

- **WebSocket**: HTML5-related protocol that provides full duplex communication channels over a single TCP connection available in modern web browsers and web servers

- **Server-Sent Events:** Strategy based on the EventSource HTML5 support that allows a server to stream messages to the connected clients

- **Forever Frame**: Strategy based on the use of a hidden iframe element, where the chunked encoding HTTP feature is used to send stream of bytes

- **Long Polling**: Basic technique that involves the opening of an HTTP connection and keeping it artificially alive to create the illusion of a persistent connection.

By default, SignalR will try to use WebSocket whenever possible and if it's not available, it will gracefully fall back on the remaining ones until it picks the best option for the current environment. So with SignalR, we can establish real-time communication from server to clients: our application can push contents to connected clients instantly, rather than having the server wait for a client requesting new data.

But what about pushing content from SQL Server to web application?

Let's assume we are developing a Web application to book flight tickets. Let us assume the tickets availability is stored in a database used by different booking terminals. These terminals have no idea about availability, and the only way for us to be aware of any change is to continuously poll the database for changes, and refresh the display. This approach has some disadvantages, such as a constant performance hit on the server even when there is no change in data.

For such applications, we need the server to take the initiative and be capable of sending information to the client when a relevant event occurs, instead of waiting for the client to request it.

But what about the database? **How can we keep our server code aware of database changes?** An easy solution is writing code that periodically re-executes a query to maintain and update the server cache, and then use SignalR to send back a message to all connected clients whenever something has changed. In software terminology, this kind of continuous checking of other programs or devices to see what state they are in, is called polling.

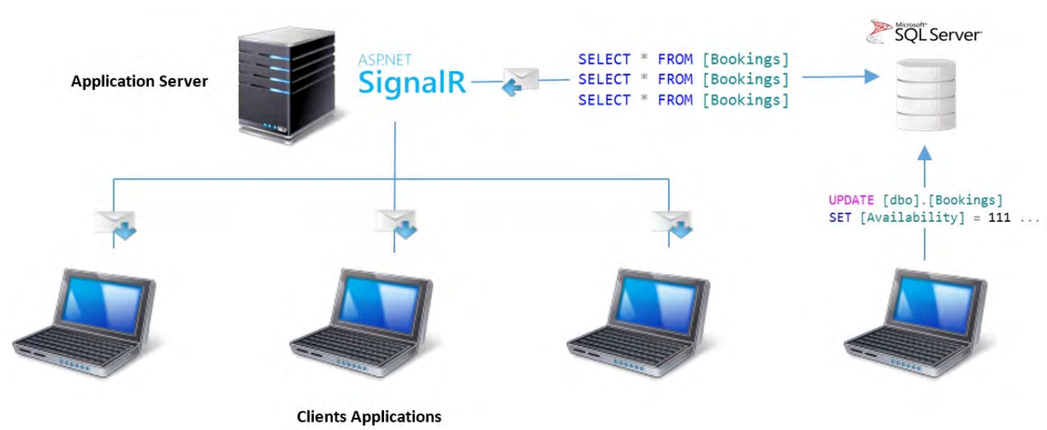


Figure 1: Server application polling database for changes and notifying client

Of course, this is not the best solution. We need something that automatically notifies our server about any record change, and thereby reduces round trips to the database. Some possible options are:

- SQL Server Service Broker.
- SqlNotificationRequest.
- SqlDependency.

Let's take a look at them.

SQL Service Broker and Queue

This is a feature of SQL Server where external or internal processes can send and receive asynchronous messages reliably by using extensions of T-SQL Data Manipulation Language (DML).

SQL Server Service Broker is used to create conversations for exchanging messages. Messages are exchanged between two parties, the destination and the initiator, and allows to transmit data and trigger processing when receiving a message.

This solution, of course, requires a good knowledge of SQL Sever needed for implementing the entire database infrastructure - that is a SQL Server Queue and a Service Broker. A server side object to listen for notifications is also needed.

SqlNotificationRequest

With SqlNotificationRequest, we are required to create our own Service Broker service and queue in SQL Server, as well as our own listener to process the sent notification accordingly. We may choose to use this lower-level class for more granular control over the notification architecture. Following steps are involved:

- Set up the Service Broker service and queue.

- Create an instance of the SqlNotificationRequest class and attach it to the SqlCommand.Notification property.

- Write a listener that retrieves and reacts to the received query notifications message.

This solution has the advantage to deal with a .NET class that receives notification about record changes, but still requires us to create a SQL Server Queue as well as a Service Broker.

SqlDependency

If we want to use query notifications without paying attention to the underlying Service Broker infrastructure, the `SqlDependency .NET` class in `System.Data` is our choice. The `SqlDependency` class represents a query notification dependency between an application and an instance of SQL Server. When we use query notifications, SQL Server provides the queue and the service object, as they are created automatically.

While SqlNotificationRequest is a lower level API, SqlDependency is a high-level implementation to access query notifications. It allows us to detect changes on the database table. In most cases, this is the simplest and most effective way to leverage the SQL Server notifications capability by managed client applications (using the .NET Framework data provider for SQL Server). In short, SqlDependency provides capabilities to our application to monitor our database table for data changes without the hassles of continuously querying the database using timers or loops.

However, the received notification doesn't tell us which record is changed, so it is necessary to execute another database query to fetch the data. This is an example showing the event handler triggered by SqlDependency due to a record change:

```
private void
SqlDependencyOnChange(object sender,
SqlNotificationEventArgs eventArgs)
{
  if (eventArgs.Info !=
  SqlNotificationInfo.Invalid)
  {
    Console.WriteLine("Notification Info:
    " + eventArgs.Info);
    Console.WriteLine("Notification
    source: " + eventArgs.Source);
    Console.WriteLine("Notification type:
    " + eventArgs.Type);
  }
}
```

As we can see, there is no information about the record that has been modified. With SqlDependency, infact the application issues a command that contains a query, and a request for notification. The application caches the results of the query or dynamic content generated from the query results. When the application receives the query notification, the application clears the cached content. The application then re-issues the query and the notification request, when the application needs the updated query results.

Wouldn't it be better if this database notification returned us updated, inserted or deleted records, avoiding us to execute a new SELECT to refresh our server cache?

SqlTableDependency for instant notifications from Database to Server

SqlTableDependency is an open source component that can create a series of database objects used to receive notifications on table record change. When any insert/update/delete operation is detected on a table, a change notification containing the record's status is sent to SqlTableDependency, thereby eliminating the need of an additional SELECT to update application's data.

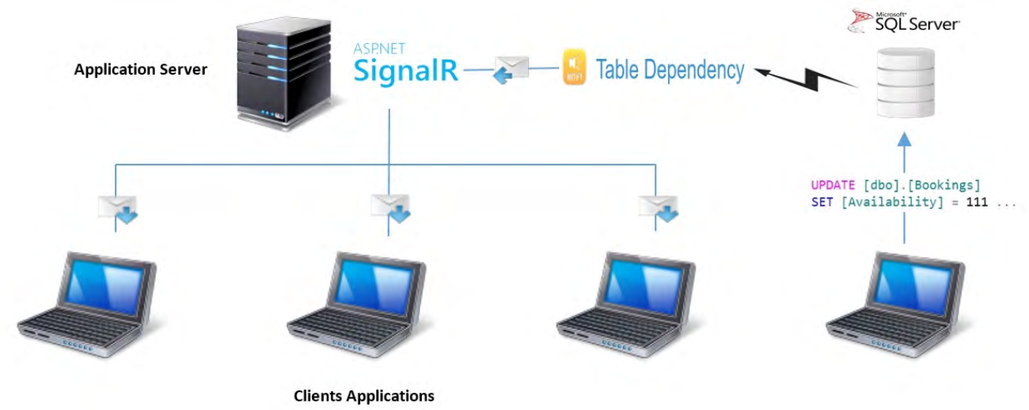


Figure 2: Notification from Database to SqlTableDependency after table record changes.

The `SqlTableDependency` class represents a notification dependency between an application and a SQL Server table. To get notifications, this component provides an on the fly low-level implementation of an infrastructure composed of a table trigger, contracts, messages, queue, service broker and a clean-up stored procedure. SqlTableDependency class provides access to notifications without knowing anything about the underlying database infrastructure. When a record change happens, this infrastructure notifies SqlTableDependency, which in turn raises a .NET event to subscribers providing the updated record values. You can read more about SqlTableDependency at https://tabledependency.codeplex.com/wikipage?title=SqlTableDependency.

Generic implementation

This event that gets invoked implements the generic <T> pattern; a subscriber receives a C# object based on our needs. Let's take a very simple example. Let's take a database table as shown in Figure 3, and define a model that maps only to table columns we are interested in:

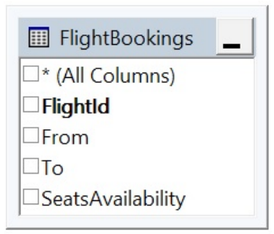


Figure 3: Database table and relative C# model.

```
public class SeatsAvailability
{
  public string From { get; set; }
  public string To { get; set; }
  public int Seats { get; set; }
}
```

After this step we can establish our dependency, by simply creating a new instance of SqlTableDependency. Here we specify the C# model used to map the column's table; and if needed, the table name as the second optional parameter.

```
var conStr = ConfigurationManager.
ConnectionStrings["connectionString"].
ConnectionString;

var mapper = new
ModelToTableMapper<SeatsAvailability>();
mapper.AddMapping(c => c.Seats,
"SeatsAvailability");

using (var tableDependency = new
SqlTableDependency<SeatsAvailability>(
   conStr,
   "FlightBookings"))
{
  tableDependency.OnChanged +=
  TableDependency_Changed;
  tableDependency.Start();

  Console.WriteLine(@"Waiting for
  receiving notifications...");
  Console.WriteLine(@"Press a key to
  stop");
  Console.ReadKey();

  tableDependency.Stop();
}
```

The `ModelToTableMapper` in this case is necessary because our C# model has a property whose name differs from the column's name. The same holds true for the second parameter passed to SqlTableDependency constructor: C# model name differs from table name. If we had adopted identical names, this configuration wouldn't be necessary. Finally we define a handler that subscribes to record changes:

```
private static void
TableDependency_Changed(object sender,
RecordChangedEventArgs
<SeatsAvailability> e)

{
  var changedEntity = e.Entity;

  Console.WriteLine(@"DML operation: " +
  e.ChangeType);
  Console.WriteLine(@"From: " +
  changedEntity.From);
  Console.WriteLine(@"To: " +
  changedEntity.To);
  Console.WriteLine(@"Seats free: " +
  changedEntity.Seats);
}
```

For every inserted, deleted or updated record, SqlTableDependency instantiates an object with properties whose values are set based on the modifications carried on by a SQL operation. To achieve this, SqlTableDependency creates the following database objects:

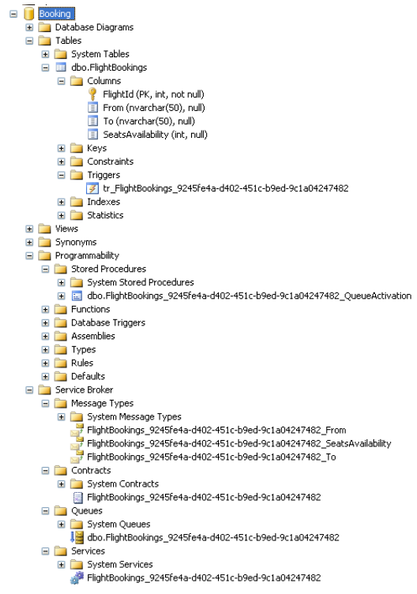


Fig.4: Database infrastructure generated from SqlTableDependency.

The most important objects to note are:

- Queue: used to store messages related to record changes
- Service broker: used to exchange info with the queue
- Table trigger: used to catch record changes on the monitored table, and prepare a message to put in the queue using the Service Broker.
- Conversation timer: This behaves as a "watch dog". When the SqlTableDependency time-out expires, Service Broker puts a message to DialogTimer on the queue. Every message inserted in the queue triggers the activated stored procedure that detects the DialogTimer message, and executes the teardown of all SQL object generated previously in order to remove unused database objects.
- Activated stored procedure: used to clean up all SqlTableDependency objects in case the application closed down without disposing SqlTableDependency.

To use SqlTableDependency, we must have SQL Server Service Broker enabled on our database.

A practical example using Knockout JS

We are going to create a web application simulating a Flight Booking system. This web application serves different client terminals that book tickets for a flight. These terminals need to stay up to date constantly to avoid overbooking. The idea is to refresh all connected terminals every time a reservation is done, by submitting a message from the server containing real time availability.

Using the described components, we can obtain a chain of notifications that starting from our database, goes to the server, and in turn goes to all terminal clients. All this without any sort of polling.

Initial settings

Create an ASP.NET MVC Web application and install the following packages:

```
PM> Install-Package Microsoft.AspNet.
SignalR
PM> Install-Package Knockoutjs
PM> Install-Package Knockout.Mapping
PM> Install-Package SqlTableDependency
```

Then initialize SignalR:

```
using Microsoft.Owin;
using Owin;

[assembly:
OwinStartup(typeof(FlightBooking.
Startup))]
namespace FlightBooking
{
  public class Startup
  {
    public void Configuration(IAppBuilder
    app)
    {
      app.MapSignalR();
    }
  }
}
```

The following are the most important points to note during this process:

- SignalR uses OWIN (Open Web Interface) for .NET as the standard interface between .NET web servers and web applications, enabling a level of indirection and abstraction that keeps our project from directly tying up to any specific hosting platform. This enables SignalR to be hosted from web applications.
- Every OWIN application must have a `Startup` class that follows specific conventions: a `Configuration()` method with the signature shown in the preceding code must be defined.
- The assembly-level attribute `OwinStartup` is used to declare that our Startup class will be used to bootstrap every OWIN-based asset contained in all the loaded assemblies.

• Inside the `Configuration()` method, we make a call to the `MapSignalR()` method in order to expose an endpoint called /signalr, which the clients will use to connect to the server.

Now we need a client. So, we customize the layout in order to include the required JavaScript libraries:

```
<!DOCTYPE html>
<html>
<head>
  <title>SignalR, Knockout JS and
  SqlTableDependency</title>
</head>
<body>
  <div class="container" style="margin-
  top: 20px">
   @RenderBody()
  </div>

  <script src="~/Scripts/jquery-
  1.10.2.js"></script>
  <script src="~/Scripts/jquery.signalR-
  2.2.0.js"></script>
  <script src="~/signalr/hubs"></script>
  <script src="~/Scripts/knockout-
  3.4.0.js"></script>
  <script src="~/Scripts/knockout.
  mapping-latest.js"></script>

  @RenderSection("scripts", required:
  false)
</body>
</html>
```

The first relevant portions of code are the first two <script> blocks, where we reference jquery and jquery.signalR as JavaScript libraries. jQuery is necessary because the SignalR JavaScript client is actually a jQuery plugin.

Then we refer to a dynamic endpoint (/signalr/hubs) exposed by the server because of the `MapSignalR` call from the Startup class. It actually generates JavaScript code on the fly according to the available Hub. In practice, the JavaScript proxies for our Hub is built on the fly by the server-side portion of SignalR as soon as this endpoint is hit, and it is sent to the client as JavaScript source code.

Hub implementation

SignalR's main goal is to deliver a real-time message over HTTP. In order to do so, SignalR comes with two distinct APIs: one called *Persistent connection*, which we can consider as the low-level API; and one called *Hub*, which built on top of the former, represents a high-level API that allows client and server to call methods on each other directly. Hubs also allow us to pass strongly typed parameters to methods, enabling model binding.

To recap our target, what we want to achieve is a notification chain that will forward this information to all connected clients, whenever any change is done to the FlightBookings table. So assuming an `update` DML operation on this table is executed, we want to broadcast messages from database, up to the clients.

So we start defining our Hub class. We are going to use it to retrieve the first set of available flight seats, and then push seats availability change from the server. This class establishes a communication channel between the server and clients:

```
[HubName("flightBookingTicker")]
public class FlightBookingHub : Hub
{
  private readonly FlightBookingService
  _flightBookingService;

  public FlightBookingHub() :
  this(FlightBookingService.Instance) {
}

  public
  FlightBookingHub(FlightBookingService
  flightBookingHub)
  {
    _flightBookingService =
    flightBookingHub;
  }

  // used to get the first result set
  concerning seats availability
  public FlightsAvailability GetAll()
  {
    return _flightBookingService.
    GetAll();
  }
}
```

Note: *The class is marked with the `HubName` attribute, which allows us to give the Hub a friendly name to be used by the clients. If we don't use the HubName attribute, the Hub name will be the same as the class name.*

Hubs are instantiated through calls. Each time a client sends a message to the server, an object of this type will be created to process the request: it means that we cannot use instance members on the hub to maintain the state. After the request is processed, the object will be eliminated, so this information would be lost. That is why we need a singleton service class to constitute the channel between the database and web application; to be able to be the listener for record modifications. For this we are going to use SqlTableDependency.

Let's create our server side "record change" listener. In its constructor, we instantiate our `SqlTableDependency` instance that will work as "record table change" listener, defining the connection string, and the table name to monitor:

```
public class FlightBookingService :
IDisposable
{
 // singleton instance
 private readonly static
 Lazy<FlightBookingService> _instance =
  new Lazy<FlightBookingService>(() =>
  new FlightBookingService(
   GlobalHost.ConnectionManager.
   GetHubContext<FlightBookingHub>().
   Clients));

   private SqlTableDependency
   <FlightAvailability>
   SqlTableDependency { get; }
   private
   IHubConnectionContext<dynamic>
   Clients { get; }

   private static connectionString =
   ConfigurationManager.
ConnectionStrings["connectionString"].
ConnectionString;

   public static FlightBookingService
Instance => _instance.Value;

   private FlightBookingService
   (IHubConnectionContext<dynamic>
   clients)
   {
    this.Clients = clients;

    // because our C# model has a
    property not matching database table
    name,
    // an explicit mapping is required
    just for this property
    var mapper = new ModelToTableMapper
    <FlightAvailability>();
    mapper.AddMapping(x =>
    x.Availability,
    "SeatsAvailability");

    // because our C# model name differs
    from table name we must specify
    table name
    this.SqlTableDependency = new
    SqlTableDependency
    <FlightAvailability>(
     connectionString,
     "FlightBookings",
     mapper);

     this.SqlTableDependency.OnChanged
     += this.TableDependency_OnChanged;
     this.SqlTableDependency.Start();
   }
```

Unlike the Hub implementation, this service class is not disposed off when the communication with the client is ended. It has to monitor record changes, and then route this info to clients using an instance of our hub implementation. This is achieved by registering an event handler on SqlTableDependency:

```
  private void TableDependency_OnChanged
  (object sender, RecordChangedEventArgs
  <FlightAvailability> e)
  {
   switch (e.ChangeType)
   {
    case ChangeType.Delete:
     this.Clients.All.
     removeFlightAvailability(e.Entity);
     break;

    case ChangeType.Insert:
     this.Clients.All.
     addFlightAvailability(e.Entity);
    break;

    case ChangeType.Update:
     this.Clients.All.
     updateFlightAvailability(e.Entity);
    break;
   }
  }
}
```

This event gives us an object populated with current table values. In case of `update` or `insert` operation, we receive the latest value; in case of *delete*, we receive the deleted record values. Within our event handler, we can notify a list of connected clients (`IHubConnectionContext<dynamic>` Clients property) about the change that happened on the database table. The JavaScript proxy will get this new availability, and using KnockoutJS, will update the UI.

Our hub also exposes a method used to retrieve the seats availability. This method will be called from our client code only once - to populate the view containing the free seats:

```
public FlightsAvailability GetAll()
{
 var flightsAvailability = new
 List<FlightAvailability>();

 using (var sqlConnection = new
 SqlConnection(connectionString))
 {
  sqlConnection.Open();
 using (var sqlCommand = sqlConnection.
 CreateCommand())
 {
  sqlCommand.CommandText = "SELECT *
  FROM [FlightBookings]";

  using (var sqlDataReader =
  sqlCommand.ExecuteReader())
  {
   while (sqlDataReader.Read())
   {
    var flightId = sqlDataReader.
    GetInt32(0);
    var from = sqlDataReader.
    GetString(1);
    var to = sqlDataReader.
    GetString(2);
    var seats = sqlDataReader.
    GetInt32(2);

    flightsAvailability.Add(new
    FlightAvailability {
     FlightId = flightId,
     From = from,
     To = to,
     Availability = seats
    });
   }
  }
 }
 return new FlightsAvailability() {
  FlightCompanyId = "field not used",
  FlightAvailability =
  flightsAvailability
 };
}
```

To complete our hub, we need to implement the `IDisposable` interface in order to destroy all database objects generated from SqlTableDependency.

```
public void Dispose()
{
 // invoke Stop() to remove all
 DB objects generated from
 SqlTableDependency
  this.SqlTableDependency.Stop();
}
```

Seats Availability View

Now it is time to create a simple controller just to render our view:

```
public class FlightBookingController :
Controller
{
  public ActionResult Index()
  {
    return View();
  }
}
```

This view takes advantage of KnockoutJS to display free seats, as well as update them every time the availability changes on the database table. Knockout is a Model-View-ViewModel (MVVM) JavaScript library that helps us to create rich, responsive user interfaces with a clean underlying data model. Any time we have sections of UI that update dynamically (e.g. changes depending on the user's actions or when an external data source changes), knockout will refresh our UI. This is achieved using the `data-bind` attribute added to DOM elements, that automatically update their values.

```
<table class="table table-striped">
 <thead style="background-color:
 silver">
  <tr>
   <th>Flight Id</th>
   <th>From</th>
   <th>To</th>
   <th>Seats Availability</th>
  </tr>
 </thead>
 <tbody data-bind="foreach: flights">
  <tr>
   <td><span data-bind="text: $data.
   flightId"></span></td>
   <td><span data-bind="text: $data.
   from"></span></td>
   <td><span data-bind="text: $data.
   to"></span></td>
   <td><span data-bind="text: $data.
   freeSeats"></span></td>
  </tr>
 </tbody>
</table>

@section Scripts {
 <script src="~/Scripts/
 flightBookingViewModels.js"></script>
 <script src="~/Scripts/
 flightBookingTicker.js"></script>
}
```

Knockout ViewModel and SignalR client code

What remains to do is to define our knockout ViewModel, to bind HTML elements with our server data. In this example, we take advantage of the Knockout Mapping plug-in. This is an alternative to manually writing your own JavaScript code that constructs a ViewModel based on data we are fetching from the server. The mapping plug-in will convert JavaScript objects or JSON strings into Knockout observable properties that will be automatically added to our ViewModel. In the end, this step results in having four properties (flightId, from, to, freeSeats) created in our ViewModel.

```
// flight ViewModel definition
function FlightBookingViewModel(flight) {
 var self = this;

 var mappingOptions = {
  key: function (data) {
   return ko.utils.
   unwrapObservable(data.flightId);
  }
 };
 ko.mapping.fromJS(flight,
 mappingOptions, self);
};

// flights view model definition
function FlightsBookingViewModel(flights)
{
 var self = this;
 var flightsBookingMappingOptions = {
  flights: {
   create: function (options) {
    return new
    FlightBookingViewModel(options.
    data);
   }
  }
 };

 self.addFlightAvailability = function
 (flight) {
  self.flights.push(new
  FlightBookingViewModel(flight));
 };

 self.updateFlightAvailability =
 function (flight) {
  var flightMappingOptions = {
   update: function (options) {
    ko.utils.arrayForEach(options.
    target, function (item) {
     if (item.flightId() === options.
     data.flightId) {
      item.freeSeats(options.data.
      freeSeats);
     }
    });
   }
  };

  ko.mapping.fromJS(flight,
  flightMappingOptions, self.flights);
 };

 self.removeFlightAvailability =
 function (flight) {
  self.flights.remove(function(item) {
   return item.flightId() === flight.
   flightId;
  });
 };

 ko.mapping.fromJS(flights,
 flightsBookingMappingOptions, self);
};
```

The main ViewModel - `FlightsBookingViewModel` - has a constructor parameter used to initialize itself with initial seats availability. Also, it exposes three

methods used to update itself, and consequently the UI

- removeFlightAvailability
- updateFlightAvailability
- addFlightAvailability

These methods will be called from the client-side hub proxy every time it is notified from our server side code using SignalR:

```
$(function () {
 var viewModel = null;
 // generate client-side hub proxy and
 then
 // add client-side hub methods that the
 server will call

 var ticker = $.connection.
 flightBookingTicker;

 // Add a client-side hub method that
 the server will call
 ticker.client.updateFlightAvailability
 = function (flight) {
  viewModel.
  updateFlightAvailability(flight);
 };

 ticker.client.addFlightAvailability =
 function (flight) {
  viewModel.
  addFlightAvailability(flight);
 };

 ticker.client.removeFlightAvailability
 = function (flight) {
  viewModel.
  removeFlightAvailability(flight);
 };

 // start the connection, load seats
 availability and set the knockout
 ViewModel
 $.connection.hub.start().
 done(function() {
  ticker.server.getAll().done(function
  (flightsBooking) {
   viewModel = new
   FlightsBookingViewModel
   (flightsBooking);
   ko.applyBindings(viewModel);
  });
 });
});
```

Our code is written inside a classic `jQuery $(...);` call, which actually ensures that it is called when the page is fully loaded. We first take a reference to our Hub, which is exposed by the `$.connection.flightBookingTicker` property generated by the dynamic endpoint.

Then we add three callback methods on flightBookingTicker hub proxy: `updateFlightAvailability`, `addFlightAvailability` and `removeFlightAvailability`, whose name and signature are matching the method that the server Hub is trying to call back. These functions will update the knockout ViewModel with the received server message every time a record is changed.

After that we call the `start()` method exposed by the `$.connection.hub` member, which performs the actual connection to our server. The `start()` call is asynchronous, and we have to make sure it has actually been completed before using any hub. That's easy because `start()` returns a promise object containing a `done()` method to which we can pass a callback function, where we put our hub-related code. Here we get the initial list of seats availability using server member (`getAll`) defined in Hub instance. Using this, we are able to call any method exposed by the Hub, creating the knockout ViewModel, and executing the applyBindings call. When this function is executed, Knockout processes both the view and the ViewModel. All data bindings in the view are executed and dynamically replaced with the data contained in the ViewModel, that in our case, is FlightBookingViewModel.

Wrapping Up

So far, instead of executing a request from client to the web application, and then the web application to the database; we are doing the reverse: sending a request from database to web application, and in turn from web application to clients. Figure 5 shows an example of notification workflow when a table record changes:

Figure 5: Notification chain when a record changes in the monitored database table

How to test

You can download the source code at bit.ly/dncm26-realtime; then create a database table with the following SQL script:

```
CREATE TABLE [dbo].[FlightBookings](
[FlightId] [int] PRIMARY KEY NOT NULL,
[From] [nvarchar](50),
[To] [nvarchar](50),
[SeatsAvailability] [int])
```

After setting the connection string, run the web application.

Initially, if the database table has no data, an empty grid will be shown, otherwise its records will be displayed. Open SQL Server Manager and add, modify or delete a record in the table. As soon as a record gets committed, we will get an immediate notification from the database to web server and, thanks to SignalR, from the web application to HTML page.

Here, with the help of Knockout JS, a single row in the grid will be modified, based on what we have received from the database.

In this article, we saw how SignalR allows bi-directional communication between server and client, and makes developing real-time web applications easy ■

Download the entire source code from GitHub at bit.ly/dncm26-realtime

About the Author

christian del bianco

Christian Del Bianco is a technology enthusiast and a C#.NET, ASP.NET MVC Software Developer. He has an experience of more than 20 years in industrial IT.

LOG ANALYTICS (OMS)

IN MICROSOFT AZURE

Vikram Pendse

Microsoft was recently announced as the leader in Cloud for PaaS, IaaS and SaaS by Gartner in their recent report . Although there is a big traction in the adoption of PaaS services, IaaS is still a key offering of Azure. Additionally, there is a huge traction in Open Source workload adoption as well. The current stats reveals that one out of three VMs in Azure is Open Source / Linux. Enterprises are not only adopting Azure as Cloud offerings, but they are also doing large scale implementations in a Hybrid Cloud environment.

Considering all these aspects, Infrastructure monitoring is one of the core activities for all IT Administrators and enterprises, as it is the backbone of the ongoing business. Log Analytics (OMS) [formerly known as "Operational Insights"] in Azure caters to all these requirements in one single service. OMS stands for Operational Management Suite. It takes care of Log Analytics, Automation, Availability and Security at one single place. The new enhanced Log Analytics also extends itself to On Premise infrastructure, Amazon (AWS) workload and Open Stack besides traditional Windows and Linux virtual infrastructure in Azure.

This article provides insights on Log Analytics in Azure.

What is Log Analytics (OMS) and who needs it?

All enterprises or Azure customers who are looking for one single dashboard to monitor logs, security and other infrastructure related crucial information, can leverage Log Analytics. The most important and unique feature of Log Analytics is that it helps you to see all details in one place not only for your Virtual Machines on Azure, but also On-Premise Machines, as well as Amazon-AWS Open Stack workloads. When we say Virtual Machines in Azure, it is in no way limited to only Windows VMs, but you can use this for Linux VMs as well. Another good part is, now you can leverage this for your Docker Containers as well. Note that Amazon-AWS and Container Log analytics is in Preview mode, and not generally available.

In this article, we will cover some important features of Log Analytics and the scenarios where it will be useful.

Introduction to Log Analytics (OMS)

What is Log Analytics (OMS)?

Log Analytics (OMS) is an Azure based service which gives you real time operational intelligence and Visualization from your Windows and Linux servers, irrespective of their location and format.

Is Log Analytics (OMS) free in Azure?

There are multiple price slabs available for Log Analytics as below (given in USD):

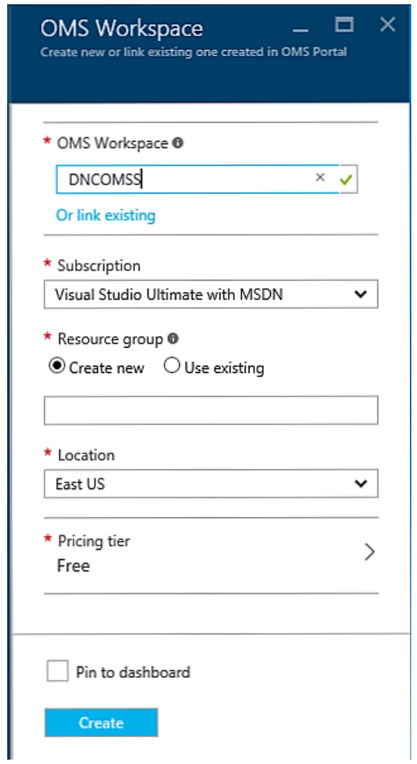
1. Free (Price: FREE, Daily Limit: 500 MB, Retention Period: 7 days)

2. Standard (Price: $2.30/GB, Daily Limit: None, Retention Period: 1 Month)

3. Premium (Price: $3.50/GB, Daily Limit: None, Retention Period: 12 Months)

How to create Log Analytics (OMS) instance in Azure Portal?

As a prerequisite you require Azure Subscription. You can click on "Browse > Log Analytics (OMS)" or can use the Global Search box at the top of Azure portal and search from there.



While creating OMS workspace, you need to give the following details:

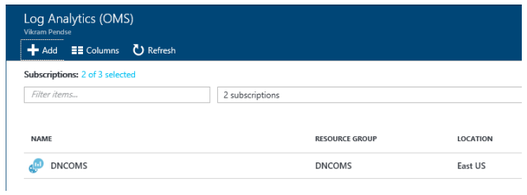
1. OMS Workspace Name (You can also link to an already created Workspace)

2. Subscription

3. Resource Group (You can either create new Resource Group or use an existing one)

4. Location (Datacenter Location)

5. Pricing Tier (By default FREE tier, you can select Standard or Premium as per your requirement)



What is the "OMS Portal" option?

Although a similar functionality like OMS was already available in a different format, it is now Cloud based (Azure) with a new name as "Log Analytics".

You can certainly perform Log Search and other various configurations from the Azure portal, however there is a different dedicated "OMS" portal which has more functionalities and options, than what you can get on the Azure portal. Hence by clicking on "OMS Portal", an altogether different portal opens up.

Note that the Azure dashboard has limited capabilities, so once you register your VMs (data source), for all other configurations, you need to rely on the OMS portal.

Adding your VMs and Machines

Users need to add existing VMs or Machines to the workspace created. Go to Settings and you can see the Virtual Machine option which enlists the Virtual Machines in your subscription. These VMs can be from different Resource Groups. Here the Quick Start dialog will show you options to connect to:

1. Azure Virtual Machines (VMs) – These VMs can be Windows or Linux as well

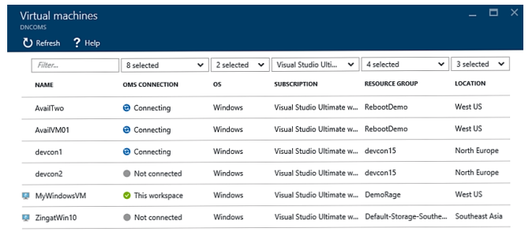
2. Storage Account Logs – Storage Account (Can be classic or Resource Manager)

3. Computers (On-Premise machines)

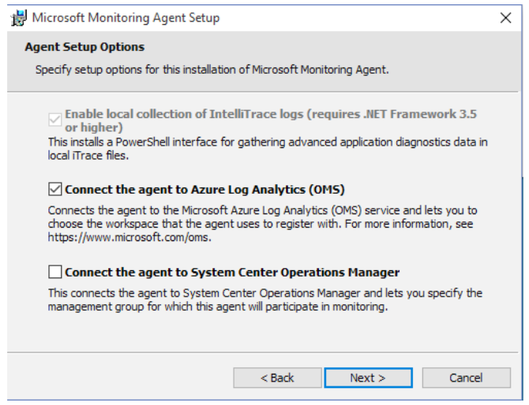
4. System Center Operations Manager (SCOM)

These components are basically treated as "Data Source" for your Log Analytics (OMS) workspace.

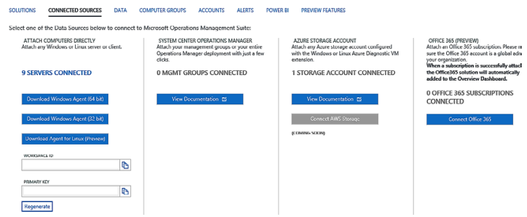
You need to select an individual VM, and need to click on "Connect"



The dashboard will give you a view of all connected VMs. To connect to your local machines or System Center Operations Manager, you need to download and install Agent setup (It is available for x64 and x86 Architecture).



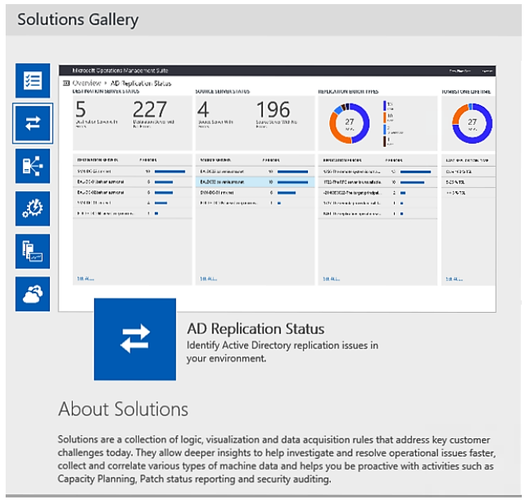
Post installation, the agent will start pushing data to the workspace. Note that while installing the Agent, it will ask you for a Workspace ID and Primary Key. You can get this info from the OMS Portal's Setting page as shown here:



You can also see the other data source options like Storage (including AWS storage which is in Preview as of this writing) and Office 365 (which is also in Preview). Once you set your Data sources, you need to start adding Solutions.

Adding Solutions to your OMS Workspace

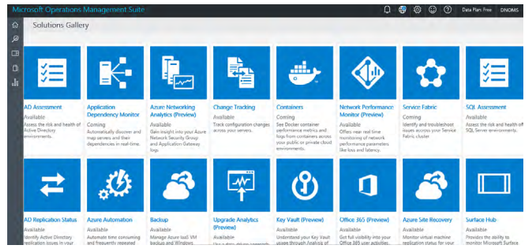
Beside Logs and Update Status Visualization, OMS gives you many useful add-in/pluggable solutions which you can add to your workspace to get every minute details of your infrastructure. Not only that, you can also customize and automate it as well, as per your requirements. Generating "Alerts" and building Power BI data sources from the collected data are some value added features.



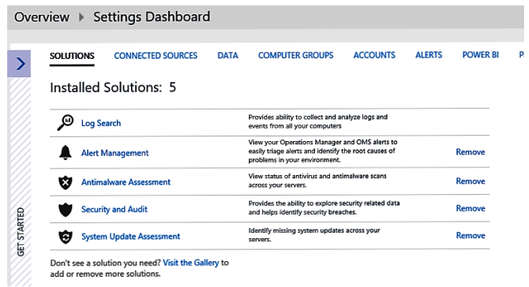
There is no specific limit on the number of solutions you can add to your workspace. There are multiple solutions available in the Gallery. Some of the commonly used solutions are as below:

1. Active Directory Assessment
2. Antimalware
3. Security
4. Alert Management
5. System Updates
6. Change Tracking
7. SQL Assessment

There are some more solutions which you can see in the Solution Gallery on OMS portal, and you can add the same to your workspace. Some of these solutions are in Preview (Not recommended for Production usage) and covers some value added solutions for Docker, Network Performance etc. Some solutions like Service Fabric and Application Dependency monitor will be made available soon. Currently it shows the status as "Coming". However, for Capacity Management, Configuration Assessment and SCOM alerts, you need SCOM Agent.



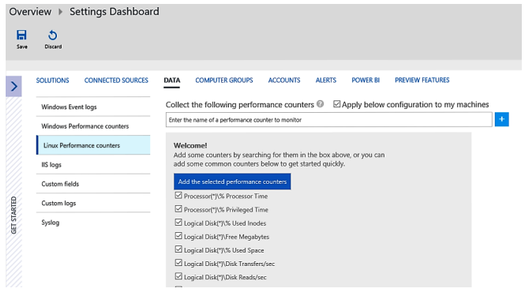
From your Settings page on OMS portal, you can see the solutions you have added, and if you want, you can remove them as well. The Solution Setting panel too has a link to the Solution Gallery.



Setting up Dashboard

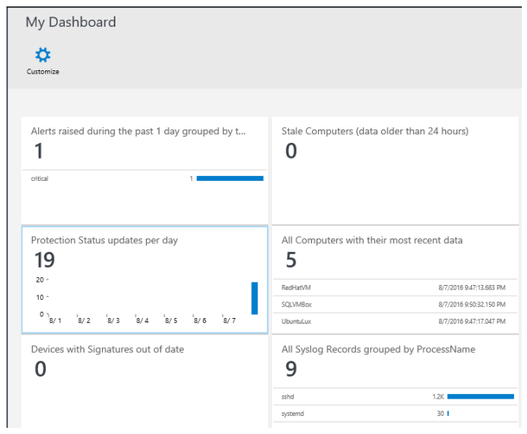
Once you create a Log Analytics instance from the Azure portal and come to the OMS portal, by default all the widgets are empty. Once you configure Data tab in Settings, you will start getting updated Visuals on the main dashboard, and your

customized dashboard as well. You can configure the log data matrix and types of logs depending on your logging and monitoring requirements. This includes Windows and Linux VMs, and Windows machines as well. You can also specify performance counter criteria/filters.

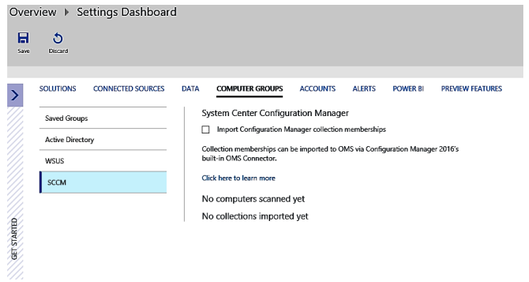


Once you configure Data (Log Matrix) in settings, you can see the visual output on the Dashboard, and can get a more granular level information from the graphs.

Personal Customized Dashboard View



In Settings, Computer Group tab allows you to collect SCCM collection Membership. You can import from Active Directory and WSUS (Windows Server Update Service) as well.

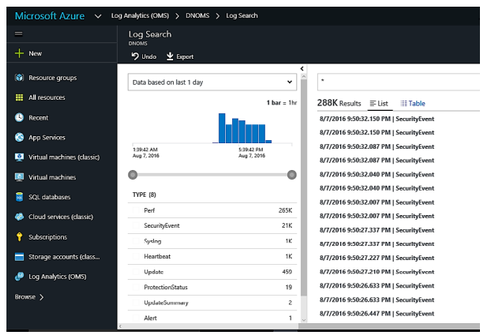


Value Add features of Log Analytics (OMS)

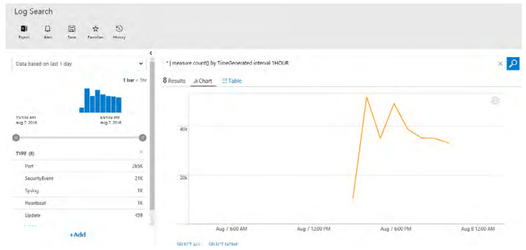
There are multiple features of OMS which in practice does a lot of logging, monitoring and automation. However, here are some key features which are heavily used, and are very useful in large enterprise scenarios.

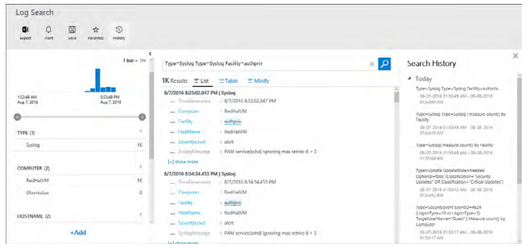
Log Search

Over a period of time, huge logs get created in OMS, and most of the time, Administrators need them handy. They also wish to get a drill down information and search facility over these massive amount of logs. OMS solves all these issues. Log Search is also available in Azure Portal along with OMS portal. However, OMS gives additional options and operations over Log Search compared to the Microsoft Azure Portal. Since the data source has multiple VMs and Machines, and there is no separate classification for Windows and Linux, hence all the Logs are at one place. However with filters, you can get more details.



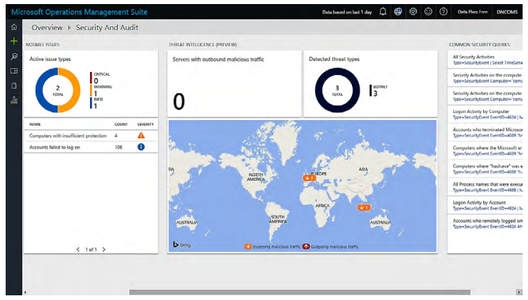
You can also export the result of Log Search to a CSV file from the New Portal. You can also Save the template and can mark it as a favorite to access it later.





Security and Audit

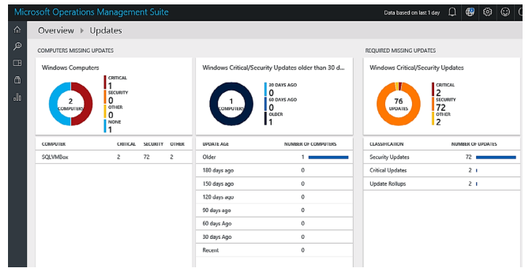
Security and Audit gives you Threat Intelligence which is very helpful to understand valid and invalid attempts to login. You can see a variety of user logins including some from the Operational Team in Microsoft. Thus it helps you to keep track of Valid/Successful Logins as well as Login failures/ Invalid attempts to login. It also keeps track of Policy changes, Change or Reset Password Attempts as well as Remote Procedure Calls (RPC) attempts.



Threat Intelligence which is currently in Preview, gives you a Map based Graphical information on the Intrusion attempts, and malicious incoming and outgoing traffic. By clicking on the push pin on the Map, you will get further drill down information like source and geography of malicious incoming traffic, Host name and IP address from where malicious incoming traffic is detected.

Updates

Windows Updates or patches is one of the most common routine, but is an important and sensitive activity done by IT team on a day to day basis. However, in a large infrastructure of VMs / Machines, it becomes a difficult task to check the status of updates.

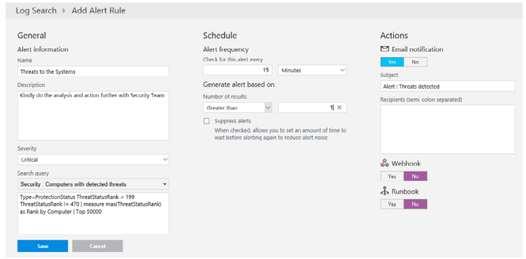


Updates provide all the information related to Windows updates in a single dashboard, and thus you can see information like Number of updates, which ones are critical, Classification of updates along with update age (update history)

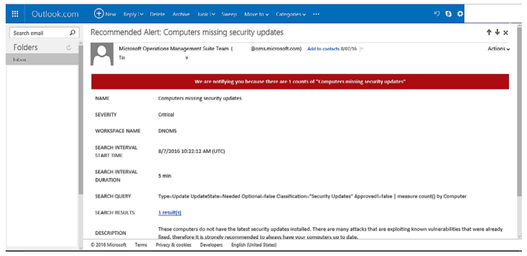
Alerts

Since the entire system should be interactive and efficient, just having all the visualization and log search is not enough. While we have seen various features of OMS till now, and the in-depth information we can get, store and utilize from it, it is also important to maintain the OMS activities running smoothly. Hence Alerts comes into the picture.

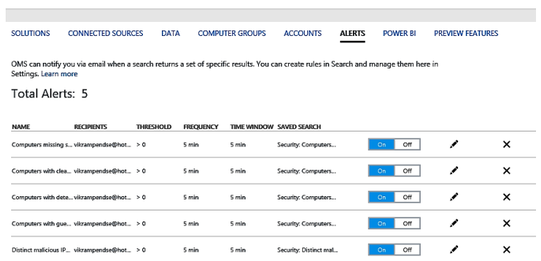
You will also see an “Alert” icon in Log Search and other options like Security and Audit as well. Here is a simple way to Add Alert for a particular Log item or value.



You can also add One or more Email recipients (usually Administrators and Support Team members) to get Alerts with necessary metadata over email.



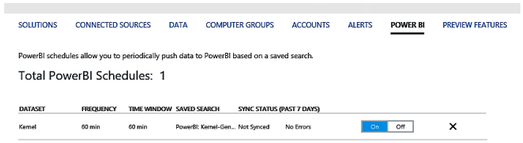
Based on the frequency set, and matrix for alert generation, you will get an alert in your inbox as shown below. You then need to do further analysis and take necessary action. You can also manage Alert Rules and do further actions/customizations from Setting page as shown below



Power BI and Preview features

Although Log Analytics (OMS) is GA (Generally available), some of the services and functionalities are still in Preview. We saw how rich the Log Analytics (OMS) tool is, and how it gathers and orchestrates all the infrastructure information into a single workspace.

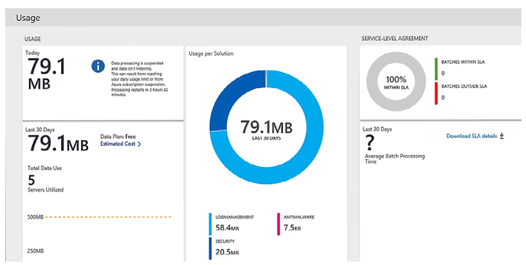
Although OMS dashboard shows Data visualization for the selective Matrix, Power BI is a generic offering for Data Visualization. Power BI provides Windows, Web and Mobile client. Log Analytics (OMS) portal helps you to generate Power BI data source, so that it can be used by many for rich data visualization.



Based on your Search query over Log data which is pushed to the Power BI data source, you can now visualize that data in your Power BI, and do further filtering and customization if required. You can enable or disable the Preview features from Preview Features tab as shown here:

Usage

As we saw there are multiple slabs available in Log Analytics (OMS) as mentioned earlier. For FREE, Standard or Premium, we can see the utilization from OMS portal as well, as seen here:



Conclusion:

Microsoft Log Analytics (OMS) is a cloud based service which helps you gain insights into details of infrastructure hosted on premises and cloud.

Log Analytics (OMS) helps you get maximum level of details of your running, ongoing infrastructure irrespective of the datacenter location availability and public clouds (including Azure and Amazon). It delivers deep level insights across datacenter and public cloud (AWS) and VMs, which consists of Windows and Linux VMs as well ■

About the Author

vikram pendse

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

A MAGAZINE FOR .NET AND JAVASCRIPT DEVS

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

We've got it all!

100K PLUS READERS

230 PLUS AWESOME ARTICLES

25 EDITIONS

FREE SUBSCRIPTION USING YOUR EMAIL

EVERY ISSUE DELIVERED

RIGHT TO YOUR INBOX

NO SPAM POLICY

SUBSCRIBE TODAY!

THANK YOU

FOR THE 26th EDITION


@damirarh


@vikrampendse


@ekapic


bit.ly/dnc-daniel


bit.ly/dnc-rahul


@yacoubmassad


bit.ly/dnc-christian


@suprotimagarwal


@saffronstroke

WRITE FOR US