**[Build Async Services with ASP.NET Web API and Entity Framework 6](https://blog.tonysneed.com/2013/03/22/async-services-asp-net-web-api-entity-framework-6/)**

Posted on [March 22, 2013](https://blog.tonysneed.com/2013/03/22/async-services-asp-net-web-api-entity-framework-6/)by [Tony Sneed](https://blog.tonysneed.com/author/tonysneed/)

If you are building web services that interact with a database, chances are they are not written in a scalable fashion.  Web services based either on [WCF](http://msdn.microsoft.com/en-us/library/dd456779.aspx), which supports both [SOAP](http://en.wikipedia.org/wiki/SOAP) and [REST](http://en.wikipedia.org/wiki/REST), or on [ASP.NET Web API](http://www.asp.net/web-api), which exclusively supports REST, use the [.NET Thread Pool](http://msdn.microsoft.com/en-us/library/ms973903.aspx) to respond to requests.  But just because services are inherently multithreaded does not make them scale when numerous requests are made simultaneously.  The very reason why threads are pooled, instead of created on the fly, is because they are an extremely expensive resource, both in terms of memory and CPU utilization.  For example, each thread consumes about 1 MB of stack space, in addition to the register set context and thread properties.

*Download the code for this post*[here*.*](http://tonysneed.com/download/AsyncWebApiDemo.zip) (Updated with code from this [post](https://blog.tonysneed.com/2013/04/01/more-fun-async-asp-net-web-api/).)

So once a thread has finished its work, it stays alive for about a minute, in the off-chance another request will arrive and the waiting thread can be used to service it.  This means that if during the time a service request is being executed, another request arrives, an additional thread will be retrieved from the Thread Pool to service the second request.  If there are no available threads, one will be created from scratch, which can take up to 500 milliseconds, during which time the request will block.  If you have numerous requests for operations that take a long time to complete, more and more threads will be created, consuming additional memory and negatively affecting your service’s performance.

The moral of the story is: *do not block the executing thread from within a service operation.*

Yet, this is precisely what happens when you perform an IO-bound task, such as when you retrieve or save data from a database, or invoke a downstream service. Here is an example of one such operation.

public class ProductsController : ApiController

{

public IEnumerable<Product> Get()

{

using (var ctx = new NorthwindSlimContext())

{

List<Product> products =

(from p in ctx.Products.Include("Category")

orderby p.ProductName

select p).ToList();

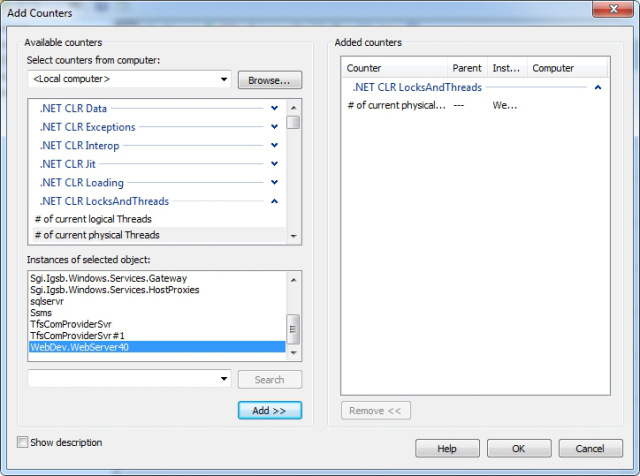
return products;

}

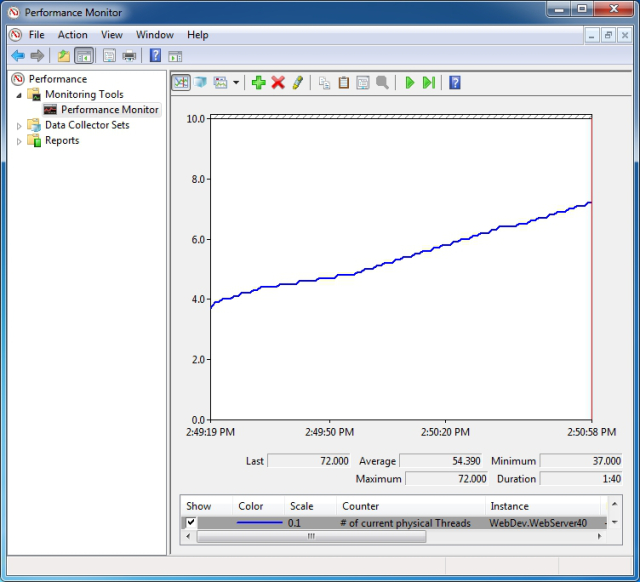
}

}

If the call to the database were to take a few seconds or more, and another call comes in (even for another method) an additional thread would need to be procured from the Thread Pool. If there are no available threads, then one would have to be created.  To get a rough picture of this phenomenon, I opened up the Windows Performance Monitor (Run, perfmon), and added a counter from the .NET CLR LocksAndThreads category, which shows the number of current physical threads for the ASP.NET Development Web Server, WebDev.WebServer.40 (aka Cassini).

[](https://tonysneed.files.wordpress.com/2013/03/curr-threads.jpg)

Running a client that issues a request every 100 milliseconds, I found that the number of current threads for the web server process increased to an average of about 55.

[](https://tonysneed.files.wordpress.com/2013/03/blocking.jpg)

Thanks to support for [asynchronous programming](http://msdn.microsoft.com/en-us/library/vstudio/hh191443.aspx) in .NET 4.5 and C# 5, it is extremely easy to write asynchronous methods for an ASP.NET Web API service.  Simply set the return type either to **Task** (if the synchronous version returns void) or to **Task<T>**, replacing T with the return type of the synchronous method.  Then from within the method, execute a *non-blocking IO-bound async operation*.  Marking the method as **async**  allows you to **await** an asynchronous operation, with the compiler converting the remainder of the methods into a continuation, or callback, that executes on another thread, usually taken from the Thread Pool.

public async **Task<IEnumerable<Product>>** Get()

{

using (var ctx = new NorthwindSlimContext())

{

List<Product> products = await

(from p in ctx.Products.Include("Category")

orderby p.ProductName

select p).**ToListAsync**();

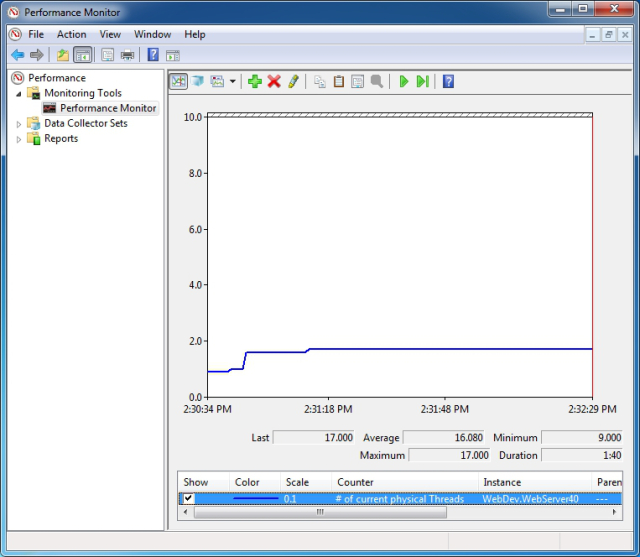
return products;

}

}

What makes this code possible is async supported added to [Entity Framework 6](http://entityframework.codeplex.com/wikipage?title=specs), available at the time of this writing as a Prerelease NuGet package.  In addition to the ToListAsync method, there are async versions of SingleOrDefault and other query methods, as well as SaveChanges – as described in the [EF6 Async Spec](http://entityframework.codeplex.com/wikipage?title=Task-based%20Asynchronous%20Pattern%20support%20in%20EF.).  What is important to emphasize here is that the async call to the database is *IO-bound versus compute-bound*.  In other words, the currently executing thread is returned immediately to the thread pool and becomes available to service other incoming requests.

When I profiled the async version of the service operation, the number of current threads hovered at around 16 — a tremendous improvement over the synchronous version!

[](https://tonysneed.files.wordpress.com/2013/03/non-blocking.jpg)

Besides database operations, another candidate for async IO would be calling downstream web services.  [HttpClient](http://www.asp.net/web-api/overview/web-api-clients/calling-a-web-api-from-a-net-client" \o "HttpClient), which comes with ASP.NET Web API, already has a task-based API for http requests, which makes it super easy to use from within an async service operation.

public async Task<string> WebServiceAsync(int id)

{

var client = new HttpClient { BaseAddress =

new Uri(WebServiceAddress) };

string json = **await** client.**GetStringAsync**(id.ToString());

var result = JsonConvert.DeserializeObject<string>(json);

return result;

}

If you are using an older web client, such as WebRequest, which has Begin and End methods, Task.Factory has a [FromAsync](http://msdn.microsoft.com/en-us/library/dd997423.aspx" \o "FromAsync) method to help you convert it to a task-based model.

public async Task<string> WebServiceFromAsync(int id)

{

var address = string.Format("{0}{1}", WebServiceAddress, id);

var webRequest = WebRequest.Create(address);

var webResponse = **await** Task<WebResponse>.Factory.**FromAsync**

(webRequest.BeginGetResponse(null, null),

webRequest.EndGetResponse);

using (var reader = new StreamReader

(webResponse.GetResponseStream()))

{

string json = reader.ReadToEnd();

var result = JsonConvert.DeserializeObject<string>(json);

return result;

}

}

If you are calling a SOAP service, [WCF 4.5](http://msdn.microsoft.com/en-us/library/dd456789.aspx) has added support for [async task-based service operations.](http://msdn.microsoft.com/en-us/library/ms734701.aspx)

[ServiceContract(Namespace = "urn:examples:services")]

public interface IValuesService

{

[OperationContract(Action = "GetValue",

ReplyAction = "GetValueResponse")]

**Task<string>** GetValue(int id);

}

The code creating a client channel looks the same, but we can now await the call to GetValue, which frees up the currently executing thread for the async service operation.

public async Task<string> SoapServiceAsync(int id)

{

var factory = new ChannelFactory<IValuesService>

(new BasicHttpBinding());

var address = new EndpointAddress(SoapServiceAddress);

var client = factory.CreateChannel(address);

using ((IDisposable)client)

{

string result = **await** client.GetValue(id);

return result;

}

}

The addition of async support for EF6 is a quantum leap forward for developers striving to write scalable services that need to perform data access. Task-based asynchrony with C# language support have been woven into both WCF and ASP.NET Web API, simplifying the effort it takes to compose asynchronous service operations.  Enjoy.