**[Parallel Programming with .NET](https://blogs.msdn.microsoft.com/pfxteam)**

All about Async/Await, System.Threading.Tasks, System.Collections.Concurrent, System.Linq, and more…

[Parallel Programming with .NET](https://blogs.msdn.microsoft.com/pfxteam/)

**Should I expose asynchronous wrappers for synchronous methods?**

*In my library, I have a method “public T Foo();”.  I’m considering exposing an asynchronous method that would simply wrap the synchronous one, e.g. “public Task<T> FooAsync() { return Task.Run(() => Foo()); }”.  Is this something you’d recommend I do in my library?*

My short answer to such a question is “no.”  But that doesn’t make for a very good blog post.  So here’s my longer, more reasoned answer…

Why Asynchrony?

There are two primary benefits I see to asynchrony: scalability and offloading (e.g. responsiveness, parallelism).  Which of these benefits matters to you is typically dictated by the kind of application you’re writing.  Most client apps care about asynchrony for offloading reasons, such as maintaining responsiveness of the UI thread, though there are certainly cases where scalability matters to a client as well (often in more technical computing / agent-based simulation workloads).  Most server apps care about asynchrony for scalability reasons, though there are cases where offloading matters, such as in achieving parallelism in back-end compute servers.

Scalability

The ability to invoke a synchronous method asynchronously does nothing for scalability, because you’re typically still consuming the same amount of resources you would have if you’d invoked it synchronously (in fact, you’re using a bit more, since there’s overhead incurred to scheduling something ), you’re just using different resources to do it, e.g. a thread from a thread pool instead of the specific thread you were executing on.  The scalability benefits touted for asynchronous implementations are achieved by decreasing the amount of resources you use, and that needs to be baked into the implementation of an asynchronous method… it’s not something achieved by wrapping around it.

As an example, consider a synchronous method Sleep that doesn’t return for N milliseconds:

public void Sleep(int millisecondsTimeout)   
{   
    Thread.Sleep(millisecondsTimeout);   
}

Now, consider the need to create an asynchronous version of this, such that the returned Task doesn’t complete for N milliseconds.  Here’s one possible implementation, simply wrapping Sleep with Task.Run to create a SleepAsync:

public Task SleepAsync(int millisecondsTimeout)   
{   
    return Task.Run(() => Sleep(millisecondsTimeout));   
}

and here’s another that doesn’t use Sleep, instead rewriting the implementation to consume fewer resources:

public Task SleepAsync(int millisecondsTimeout)   
{   
    TaskCompletionSource<bool> tcs = null;   
    var t = new Timer(delegate { tcs.TrySetResult(true); }, null, –1, -1);   
    tcs = new TaskCompletionSource<bool>(t);   
    t.Change(millisecondsTimeout, -1);   
    return tcs.Task;   
}

Both of these implementations provide the same basic behavior, both completing the returned task after the timeout has expired.  However, from a scalability perspective, the latter is much more scalable.  The former implementation consumes a thread from the thread pool for the duration of the wait time, whereas the latter simply relies on an efficient timer to signal the Task when the duration has expired.

Offloading

The ability to invoke a synchronous method asynchronously can be very useful for responsiveness, as it allows you to offload long-running operations to a different thread.  This isn’t about *how many* resources you consume, but rather is about *which* resources you consume.  For example, in a UI app, the specific thread handling pumping UI messages is “more valuable” for the user experience than are other threads, such as those in the ThreadPool.  So, asynchronously offloading the invocation of a method from the UI thread to a ThreadPool thread allows us to use the less valuable resources.  This kind of offloading does not require modification to the implementation of the operation being offloaded, such that the responsiveness benefits can be achieved via wrapping.

The ability to invoke a synchronous method asynchronously can also be very useful not just for changing threads, but more generally for escaping the current context.  For example, sometimes we need to invoke some user-provided code but we’re not in a good place to do it (or we’re not sure if we are).  Maybe a lock is held higher up the stack and we don’t want to invoke the user code while holding the lock.  Maybe we suspect we’re being invoked by some user code that doesn’t expect us to take a very long time. Rather than invoking the operation synchronously and as part of whatever is higher-up on the call stack, we can invoke the functionality asynchronously.

The ability to invoke a synchronous method asynchronously is also important for parallelism.  Parallel programming is all about taking a single problem and splitting it up into sub-problems that can each be processed concurrently.  If you were to split a problem into sub-problems but then process each sub-problem serially, you wouldn’t get any parallelism, as the entire problem would be processed on a single thread.  If, instead, you offload a sub-problem to another thread via asynchronous invocation, you can then process the sub-problems concurrently.  As with responsiveness, this kind of offloading does not require modification to the implementation of the operation being offloaded, such that parallelism benefits can be achieved via wrapping.

What does this have to do with my question?

Let’s get back to the core question: should we expose an asynchronous entry point for a method that’s actually synchronous?  The stance we’ve taken in .NET 4.5 with the [Task-based Async Pattern](http://www.microsoft.com/download/en/details.aspx?id=19957) is a staunch “no.”

Note that in my previous discussion of scalability and ofloading, I called out that the way to achieve scalability benefits is by modifying the actual implementation, whereas offloading can be achieved by wrapping and doesn’t require modifying the actual implementation.  That’s the key.  Wrapping a synchronous method with a simple asynchronous façade does not yield any scalability benefits.  And in such cases, by exposing only the synchronous method, you get some nice benefits, e.g.

* Surface area of your library is reduced.  This means less cost to you (development, testing, maintenance, documentation, etc.).  It also means that your user’s choices are simplified.  While some choice is typically a good thing, too much choice often leads to lost productivity.  If I as a user am constantly faced with both a synchronous and an asynchronous method for the same operation, I constantly need to evaluate which of the pairs is the right one for me to use in each situation.
* Your users will know whether there are actually scalability benefits to using exposed asynchronous APIs, since by definition then only APIs that benefit scalability are exposed asynchronously.
* The choice of whether to invoke the synchronous method asynchronously is left up to the developer. Async wrappers around sync methods have overhead (e.g. allocating the object to represent the operation, context switches, synchronization around queues, etc.).  If, for example, your customer is writing a high-throughput server app, they don’t want to spend cycles on overhead that’s not actually benefiting them in any way, so they can just invoke the synchronous method.  If both the synchronous method and an asynchronous wrapper around it are exposed, the developer is then faced with thinking they should invoke the asynchronous version for scalability reasons, but in reality will actually be hurting their throughput by paying for the additional offloading overhead without the scalability benefits.

If a developer needs to achieve better scalability, they can use any async APIs exposed, and they don’t have to pay additional overhead for invoking a faux async API.  If a developer needs to achieve responsiveness or parallelism with synchronous APIs, they can simply wrap the invocation with a method like Task.Run.

The idea of exposing “async over sync” wrappers is also a very slippery slope, which taken to the extreme could result in every single method being exposed in both synchronous and asynchronous forms.  Many of the folks that ask me about this practice are considering exposing async wrappers for long-running CPU-bound operations.  The intention is a good one: help with responsiveness.  But as called out, responsiveness can easily be achieved by the consumer of the API, and the consumer can actually do so at the right level of chunkiness, rather than for each chatty individual operation.  Further, defining what operations could be long-running is surprisingly difficult.  The time complexity of many methods often varies significantly.

Consider, for example, a simple method like Dictionary<TKey,TValue>.Add(TKey,TValue).  This is a really fast method, right?  Typically, yes, but remember how dictionary works: it needs to hash the key in order to find the right bucket to put it into, and it needs to check for equality of the key with other entries already in the bucket.  Those hashing and equality checks can result in calls to user code, and who knows what those operations do or how long they take.  Should every method on dictionary have an asynchronous wrapper exposed? That’s obviously an extreme example, but there are simpler ones, like Regex.  The complexity of the regular expression pattern provided to Regex as well as the nature and size of the input string can have significant impact on the running time of matching with Regex, so much so that Regex now supports optional timeouts… should every method on Regex have an asynchronous equivalent?  I really hope not.

Guideline

This has all been a very long-winded way of saying that I believe the only asynchronous methods that should be exposed are those that have scalability benefits over their synchronous counterparts.  Asynchronous methods should not be exposed purely for the purpose of offloading: such benefits can easily be achieved by the consumer of synchronous methods using functionality specifically geared towards working with synchronous methods asynchronously, e.g. Task.Run.

Of course, there are exceptions to this, and you can witness a few such exceptions in .NET 4.5.

For example, the abstract base Stream type provides ReadAsync and WriteAsync methods.  In most cases, derived Stream implementations work with data sources that aren’t in-memory, and thus involve disk I/O or network I/O of some kind.  As such, it’s very likely that derived implementations will be able to provide implementations of ReadAsync and WriteAsync that utilize asynchronous I/O rather than synchronous I/O that blocks threads, and thus there are scalability benefits to having ReadAsync and WriteAsync methods.  Further, we want to be able to work with these methods polymorphically, without regard for the concrete stream type, so we want to have these as virtual methods on the base class.  However, the base class doesn’t know how to implement these base implementations with asynchronous I/O, so the best it can do is provide asynchronous wrappers for the synchronous Read and Write methods (in actuality, ReadAsync and WriteAsync actually wrap BeginRead/EndRead and BeginWrite/EndWrite, respectively, which if not overridden will in turn wrap the synchronous Read and Write methods with an equivalent of Task.Run).

Another example in the same vein is TextReader, providing methods like ReadToEndAsync, which on the base class simply uses a Task to wrap an invocation of TextReader.ReadToEnd.  The expectation, however, is that the derived types developers actually use will override ReadToEndAsync to provide implementations that benefit scalability, such as StreamReader’s ReadToEndAsync method which utilizes Stream.ReadAsync.