Should I expose synchronous wrappers for asynchronous methods?

Stephen Toub - MSFT

April 13th, 2012

In a previous post Should I expose asynchronous wrappers for synchronous methods?, I discussed "async over sync," the notion of using synchronous functionality asynchronously and the benefits that doing so may or may not yield. The other direction of "sync over async" is also interesting to explore.

Avoid Exposing Synchronous Wrappers for Asynchronous Implementations

In my discussion of "async over sync," I strongly suggested that if you have an API which internally is implemented synchronously, you should not expose an asynchronous counterpart that simply wraps the synchronous method in Task.Run. Rather, if a consumer of your library wants to use the synchronous method asynchronously, they can do so on their own. The post outlines a variety of benefits to this approach.

Similar guidance applies in the reverse direction as well: if you expose an asynchronous endpoint from your library, avoid exposing a synchronous method that just wraps the asynchronous implementation. Doing so hides from the consumer the true nature of the implementation, and it should be left up to the consumer how they want to consume the implementation. If the consumer chooses to block waiting for the asynchronous implementation to complete, that's up to the caller, and they can do so with their eyes wide open.

This is even more important for the "**s**ync over **as**ync" case than it is for the "**as**ync over **s**ync" case, because for "**s**ync over **as**ync," it can lead to significant problems with the application, such as hangs.

Basic Wrapping

To understand why, let's examine what it looks like to wrap an asynchronous method with a synchronous one. How this is done, of course, depends on the asynchronous pattern employed by the method.

If I have an APM implementation:

```
public IAsyncResult BeginFoo(…, AsyncCallback callback, object state);
public TResult EndFoo(IAsyncResult asyncResult);
```

then a simple synchronous wrapper might look like:

```
public TResult Foo(…)
{
    IAsyncResult ar = BeginFoo(…, null, null);
    return EndFoo(ar);
}
```

In the APM pattern, if the EndXx method is provided with an IAsyncResult from the BeginXx method, and the IAsyncResult's IsCompleted returns false, the call to EndXx needs to block until the operation is completed. Hence, we call BeginXx and then immediately call EndXx to block until the asynchronous operation has completed, at which point EndXx will be able to properly return the result of the operation or propagate its exception.

A more modern asynchronous implementation would use the Task-based Asynchronous Pattern:

```
public Task<TResult> FooAsync(…);
```

and for it a simple synchronous wrapper might look like:

```
public TResult Foo(…)
{
    return FooAsync(…).Result;
}
```

Here we're accessing the return Task's Result, which will wait for the result to be available before returning it.

Real-World Example

These seem like reasonable wrappers, and depending on the situation, they may be… but of course they may not be.

Let's say for sake of example that BeginFoo/EndFoo and FooAsync are all good asynchronous implementations, using async I/O under the covers such that no threading resources are consumed for the vast majority of their execution; only at the very end of their execution do they need to do a small amount of processing in order to handle the results received from the asynchronous I/O, and they'll do this processing internally by queuing the work to the ThreadPool. This is quite reasonable, and a very common phenomenon in asynchronous implementations. And, let's say that someone has set an upper limit on the number of threads in the ThreadPool to 25 using ThreadPool.SetMaxThreads or via a configuration flag.

Now, you call the synchronous Foo wrapper method from a ThreadPool thread, e.g. Task.Run(()=>Foo()). What happens? Foo is invoked, it kicks off the asynchronous operation, and then immediately blocks the ThreadPool thread waiting for the operation to complete. When the async I/O eventually completes, it queues a work item to the ThreadPool in order to complete the processing, and that work item will be handled by one of the 24 free threads (1 of the threads is currently blocked in Foo). Great.

But now let's say that instead of queuing one task to call Foo, you queue 25 tasks to call Foo. Each of the 25 threads in the ThreadPool will pick up a task, and invoke Foo. Each of those calls will start some asynchronous I/O and will block until the work completes. The async I/O for all of the 25 Foo calls will complete and result in the final processing work items getting queued to the pool. But the pool threads are now all blocked waiting for the calls to Foo to complete, which won't happen until the queued work items get processed, which won't happen until threads become available, which won't happen until the calls to Foo complete. Deadlock!

If this seems far-fetched, know that this exact situation was quite common in .NET 1.x with a popular Framework method: HttpWebRequest.GetResponse. GetResponse was implemented almost exactly like Foo is in our example wrapper: it called the asynchronous BeginGetResponse, and then turned around and immediately waited on it using EndGetResponse. Further, in .NET 1.x, the maximum number of threads in the ThreadPool was, by default, a low number like 25. As a result, developers using HttpWebRequest.GetResponse would find themselves in deadlock territory; as a stop-gap measure, the implementation had a heuristic to see how many threads were available in the ThreadPool, and would throw an exception if it appeared a deadlock would result (http://support.microsoft.com/kb/815637). In .NET 2.0, HttpWebRequest.GetResponse was fixed to be a truly synchronous implementation, rather than wrapping BeginGetResponse/EndGetResponse.

UI

The previous example may seem esoteric, but there is a scheduler in most apps today that has an extremely limited number of threads and that can easily be deadlocked with a situation like this: the UI. The blog post Await, and UI, and deadlocks! Oh my! describes how you can deadlock the UI thread in a similar situation.

Imagine the following:

```
private void button1_Click(object sender, EventArgs e)
{
    Delay(15);
}

private void Delay(int milliseconds)
{
    DelayAsync(milliseconds).Wait();
}

private async Task DelayAsync(int milliseconds)
{
    await Task.Delay(milliseconds);
}
```

While this may look innocent, invoking Delay from the UI thread like this is almost certainly going to deadlock. By default, await'ing a Task will post the remainder of the method's invocation back to the SynchronizationContext from which the await was issued (even if that "remainder" is just the completing of the method). Here, the UI thread calls Delay, which calls DelayAsync, which awaits a Task that won't complete for a few milliseconds. The UI thread then synchronously blocks in the call to Wait(), waiting for the Task returned from DelayAsync to complete. A few milliseconds later, the task returned from Task.Delay completes, and the await causes the remainder of DelayAsync's execution to be posted back to the UI's SynchronizationContext. That continuation needs to execute so that the Task returned from DelayAsync may complete, but that continuation can't execute until the call to button1_Click returns, which won't happen until the Task returned from DelayAsync completes, which won't happen until the continuation executes. Deadlock!

Had Delay instead been invoked from a console app, or from a unit test harness that didn't have a similarly constrictive SynchronizationContext, everything likely would have completed successfully. This highlights the danger in such sync over async behavior: its success can be significantly impacted by the environment in which it's used. That's a core reason why it should be left up to the caller to decide whether to do such blocking, as they are much more aware of the environment in which they're operating than is the library they're calling.

Async All the Way Down

The point here is that you need to be extremely careful when wrapping asynchronous APIs as synchronous ones, as if you're not careful, you could run into real problems. If you ever find yourself thinking you need to call an asynchronous method from something that's expecting a synchronous invocation (e.g. you're implementing an interface which has a synchronous method on it, but in order to implement that interface you need to use functionality that's only exposed asynchronously), first make sure it's truly, truly necessary; while it may seem more expedient to wrap "sync over async" rather than to re-plumb this or that code path to be asynchronous from top to bottom, the refactoring is often the better long-term solution if it's possible.

What if I really do need "sync over async"?

In some cases, for whatever reason, you may actually need to do "sync over async." In such cases, there are some things you can do to ease the pain.

Test in Multiple Environments

Make sure you test your wrapper in a variety of environments: from a UI thread, from the ThreadPool, under stress on the ThreadPool with a low maximum set on the number of allowed threads, etc.

Avoid Unnecessary Marshaling

If at all possible, make sure the async implementation you're calling doesn't need the blocked thread in order to complete the operation (that way, you can just use normal blocking mechanisms to wait synchronously for the asynchronous work to complete elsewhere). In the case of async/await, this typically means making sure that any awaits inside of the asynchronous implementation you're calling are using ConfigureAwait(false) on all await points; this will prevent the await from trying to marshal back to the current SynchronizationContext. As a library implementer, it's a best practice to always use ConfigureAwait(false) on all of your awaits, unless you have a specific reason not to; this is good not only to help avoid these kinds of deadlock problems, but also for performance, as it avoids unnecessary marshaling costs.

Offload to Another Thread

Consider offloading to a different thread, which is typically possible unless the method you're invoking has some kind of thread affinity (e.g. it accesses UI controls). Let's say you have methods like the following:

```
int Sync() // caller needs this to return synchronously
{
    return Library.FooAsync().Result;
}

// in a library; uses await without ConfigureAwait(false)
public static Task<int> FooAsync();
```

As described above, FooAsync is using await without a ConfigureAwait(false), and as you don't own the code, you're unable to fix that. Further, the Sync method you're implementing is being called from the UI thread, or more generally from a context prone to deadlocking due to a limited number of participating threads (in the case of the UI, that limited number is one). Solution? Ensure that the await in the FooAsync method doesn't find a context to marshal back to. The simplest way to do that is to invoke the asynchronous work from the ThreadPool, such as by wrapping the invocation in a Task.Run, e.g.

```
int Sync()
{
    return Task.Run(() => Library.FooAsync()).Result;
}
```

FooAsync will now be invoked on the ThreadPool, where there won't be a SynchronizationContext, and the continuations used inside of FooAsync won't be forced back to the thread that's invoking Sync().

Consider a Nested Message Loop

If you do find that you need to invoke an asynchronous method and "block" waiting for it (to satisfy a synchronous contract), and if that method marshals back to the current thread, and if changing the method's implementation won't work (e.g. because you don't have the ability to modify it), and if offloading won't work (e.g. because the asynchronous method is thread affine), you're in a tight spot. One saving grace may be that there are multiple ways "blocking" behavior can be achieved.

The simplest and most general approach to blocking is to just to wait on a synchronization primitive. This is typically what happens when you call an EndXx method on the IAsyncResult returned from the BeginXx method, with the EndXx method using the IAsyncResult's AsyncWaitHandle to wait until the IAsyncResult transitions to a completed state. The primitive's waiting implementation may itself have some smarts that could help in some limited situations. For example, if you Wait on a Task that is backed by a delegate (e.g. it was created with Task.Run, Task.Factory.StartNew, etc.) and that's waiting to run, it's possible that TPL will decide to execute the Task on the current thread as part of the Wait call, a behavior we often describe as inlining. Typically, however, such specialized behaviors only apply in corner cases.

If you're blocking the UI thread, it's likely that the UI's message loop won't be pumping all the messages necessary to keep the UI responsive and to eventually process the message that would allow the thing you're waiting on to complete and to avoid the deadlock. For such cases, as a last ditch effort, you could consider "blocking" via a "nested message loop".

A nested message loop is just what it sounds like. If, for example, you're executing as part of a button click event handler, you're being called from a message loop that's dispatching the processing for a button click message it received. If that button click handler then itself synchronously spins up its own message loop, that inner loop is nested within the outer one. Here, for example, is a nested message loop in Windows Forms that spins waiting for a task to complete; the UI will remain responsive, since we're forcing messages in the queue to be drained via the repeated calls to Application.DoEvents (warning: this is just for demonstrative purposes, and I'm not recommending you do this… see "Keeping your UI Responsive and the Dangers of Application.DoEvents" for more details):

```
static T WaitWithNestedMessageLoop<T>(this Task<T> task)
{
    while (!task.IsCompleted)
        Application.DoEvents();
    return task.Result;
}
```

Some UI frameworks have built-in support for nested message loops, and in a more efficient and robust manner than the ad-hoc spinning with DoEvents I've done above. WPF, for example, uses the DispatchFrame and Dispatcher.PushFrame constructs to process a nested message loop. In the following example, the nested loop will exit when the task completes and sets the DispatcherFrame's Continue property to false.

```
static T WaitWithNestedMessageLoop<T>(this Task<T> task)
{
    var nested = new DispatcherFrame();
    task.ContinueWith(_ => nested.Continue = false, TaskScheduler.Default);
    Dispatcher.PushFrame(nested);
    return task.Result;
}
```

Even when a framework has a built-in notion of nested message loops, they're far from an ideal solution. It's much better to asynchronously wait whenever possible, allowing the higher-level message loop to handle all of the processing. This is just something you can consider if you're truly in a bind.

Conclusion

Getting back to the original question for this blog post: should you expose synchronous wrappers for asynchronous methods? Please try hard not to; leave it up to the consumer of the your method to do so if they must. You should consider the possibility that someone will consume your functionality synchronously, so code accordingly, with as few dependences on the threading environment as possible. And at the very least, if you must expose a synchronous wrapper that will just block waiting for an asynchronous implementation to complete, please document it accordingly so that a consumer knows what they're consuming and can plan accordingly. If you must consume an asynchronous method in a synchronous manner, do so with your eyes wide open, being careful to think through potentially problematic situations like deadlocks.

Stephen Toub - MSFT Partner Software Engineer, .NET
Follow