When you use async/await, there is no guarantee that the method you call when you do await FooAsync() will actually run asynchronously. The internal implementation is free to return using a completely synchronous path.

If you're making an API where it's critical that you don't block and you run some code asynchronously, and there's a chance that the called method will run synchronously (effectively blocking), using await Task.Yield() will force your method to be asynchronous, and return control at that point. The rest of the code will execute at a later time (at which point, it still may run synchronously) on the current context.

This can also be useful if you make an asynchronous method that requires some "long running" initialization, ie:

private async void button\_Click(object sender, EventArgs e)

{

await Task.Yield(); // Make us async right away

var data = ExecuteFooOnUIThread(); // This will run on the UI thread at some point later

await UseDataAsync(data);

}

Without the Task.Yield() call, the method will execute synchronously all the way up to the first call to await.