

09/04/2015 • 18 minutes to read

In this article

- Avoid Async Void
- Async All the Way
- Configure Context
- Know Your Tools

March 2013

Volume 28 Number 03

# Async/Await - Best Practices in Asynchronous Programming

By [Stephen Cleary](#) | March 2013

These days there’s a wealth of information about the new async and await support in the Microsoft .NET Framework 4.5. This article is intended as a “second step” in learning asynchronous programming; I assume that you’ve read at least one introductory article about it. This article presents nothing new, as the same advice can be found online in sources such as Stack Overflow, MSDN forums and the async/await FAQ. This article just highlights a few best practices that can get lost in the avalanche of available documentation.

The best practices in this article are more what you’d call “guidelines” than actual rules. There are exceptions to each of these guidelines. I’ll explain the reasoning behind each guideline so that it’s clear when it does and does not apply. The guidelines are summarized in **Figure 1**; I’ll discuss each in the following sections.

**Figure 1 Summary of Asynchronous Programming Guidelines**

Name	Description	Exceptions
Avoid async void	Prefer async Task methods over async void methods	Event handlers
Async all the way	Don’t mix blocking and async code	Console main method

Configure  
context

Use ConfigureAwait(false) when you can

Methods that require  
context

## Avoid Async Void

There are three possible return types for async methods: Task, Task<T> and void, but the natural return types for async methods are just Task and Task<T>. When converting from synchronous to asynchronous code, any method returning a type T becomes an async method returning Task<T>, and any method returning void becomes an async method returning Task. The following code snippet illustrates a synchronous void-returning method and its asynchronous equivalent:

XML


 Copy

```
void MyMethod()  
{  
    // Do synchronous work.  
    Thread.Sleep(1000);  
}  
async Task MyMethodAsync()  
{  
    // Do asynchronous work.  
    await Task.Delay(1000);  
}
```

Void-returning async methods have a specific purpose: to make asynchronous event handlers possible. It is possible to have an event handler that returns some actual type, but that doesn't work well with the language; invoking an event handler that returns a type is very awkward, and the notion of an event handler actually returning something doesn't make much sense. Event handlers naturally return void, so async methods return void so that you can have an asynchronous event handler. However, some semantics of an async void method are subtly different than the semantics of an async Task or async Task<T> method.

Async void methods have different error-handling semantics. When an exception is thrown out of an async Task or async Task<T> method, that exception is captured and placed on the Task object. With async void methods, there is no Task object, so any exceptions thrown out of an async void method will be raised directly on the SynchronizationContext that was active when the async void method started. **Figure 2** illustrates that exceptions thrown from async void methods can't be caught naturally.

Figure 2 Exceptions from an Async Void Method Can't Be Caught with Catch

XML	 Copy
<pre>private async void ThrowExceptionAsync() {     throw new InvalidOperationException(); } public void AsyncVoidExceptions_CannotBeCaughtByCatch() {     try     {         ThrowExceptionAsync();     }     catch (Exception)     {         // The exception is never caught here!         throw;     } }</pre>	

These exceptions can be observed using `AppDomain.UnhandledException` or a similar catch-all event for GUI/ASP.NET applications, but using those events for regular exception handling is a recipe for unmaintainability.

Async void methods have different composing semantics. Async methods returning `Task` or `Task<T>` can be easily composed using `await`, `Task.WhenAny`, `Task.WhenAll` and so on. Async methods returning void don't provide an easy way to notify the calling code that they've completed. It's easy to start several async void methods, but it's not easy to determine when they've finished. Async void methods will notify their `SynchronizationContext` when they start and finish, but a custom `SynchronizationContext` is a complex solution for regular application code.

Async void methods are difficult to test. Because of the differences in error handling and composing, it's difficult to write unit tests that call async void methods. The MSTest asynchronous testing support only works for async methods returning `Task` or `Task<T>`. It's possible to install a `SynchronizationContext` that detects when all async void methods have completed and collects any exceptions, but it's much easier to just make the async void methods return `Task` instead.

It's clear that async void methods have several disadvantages compared to async `Task` methods, but they're quite useful in one particular case: asynchronous event handlers. The differences in semantics make sense for asynchronous event handlers. They raise their exceptions directly on the `SynchronizationContext`, which is similar to how synchronous

event handlers behave. Synchronous event handlers are usually private, so they can't be composed or directly tested. An approach I like to take is to minimize the code in my asynchronous event handler—for example, have it await an async Task method that contains the actual logic. The following code illustrates this approach, using async void methods for event handlers without sacrificing testability:

XML

 Copy

```
private async void button1_Click(object sender, EventArgs e)
{
    await Button1ClickAsync();
}
public async Task Button1ClickAsync()
{
    // Do asynchronous work.
    await Task.Delay(1000);
}
```

Async void methods can wreak havoc if the caller isn't expecting them to be async. When the return type is Task, the caller knows it's dealing with a future operation; when the return type is void, the caller might assume the method is complete by the time it returns. This problem can crop up in many unexpected ways. It's usually wrong to provide an async implementation (or override) of a void-returning method on an interface (or base class). Some events also assume that their handlers are complete when they return. One subtle trap is passing an async lambda to a method taking an Action parameter; in this case, the async lambda returns void and inherits all the problems of async void methods. As a general rule, async lambdas should only be used if they're converted to a delegate type that returns Task (for example, Func<Task>).

To summarize this first guideline, you should prefer async Task to async void. Async Task methods enable easier error-handling, composability and testability. The exception to this guideline is asynchronous event handlers, which must return void. This exception includes methods that are logically event handlers even if they're not literally event handlers (for example, ICommand.Execute implementations).

## Async All the Way


Asynchronous code reminds me of the story of a fellow who mentioned that the world was suspended in space and was immediately challenged by an elderly lady claiming that the world rested on the back of a giant turtle. When the man enquired what the turtle was standing on, the lady replied, "You're very clever, young man, but it's turtles all the way

down!" As you convert synchronous code to asynchronous code, you'll find that it works best if asynchronous code calls and is called by other asynchronous code—all the way down (or "up," if you prefer). Others have also noticed the spreading behavior of asynchronous programming and have called it "contagious" or compared it to a zombie virus. Whether turtles or zombies, it's definitely true that asynchronous code tends to drive surrounding code to also be asynchronous. This behavior is inherent in all types of asynchronous programming, not just the new `async/await` keywords.

"Async all the way" means that you shouldn't mix synchronous and asynchronous code without carefully considering the consequences. In particular, it's usually a bad idea to block on async code by calling `Task.Wait` or `Task.Result`. This is an especially common problem for programmers who are "dipping their toes" into asynchronous programming, converting just a small part of their application and wrapping it in a synchronous API so the rest of the application is isolated from the changes. Unfortunately, they run into problems with deadlocks. After answering many async-related questions on the MSDN forums, Stack Overflow and e-mail, I can say this is by far the most-asked question by async newcomers once they learn the basics: "Why does my partially async code deadlock?"

**Figure 3** shows a simple example where one method blocks on the result of an async method. This code will work just fine in a console application but will deadlock when called from a GUI or ASP.NET context. This behavior can be confusing, especially considering that stepping through the debugger implies that it's the `await` that never completes. The actual cause of the deadlock is further up the call stack when `Task.Wait` is called.

Figure 3 A Common Deadlock Problem When Blocking on Async Code


XML	 Copy
<pre>public static class DeadlockDemo {     private static async Task DelayAsync()     {         await Task.Delay(1000);     }     // This method causes a deadlock when called in a GUI or ASP.NET context.     public static void Test()     {         // Start the delay.         var delayTask = DelayAsync();         // Wait for the delay to complete.         delayTask.Wait();     } }</pre>	

The root cause of this deadlock is due to the way `await` handles contexts. By default, when an incomplete Task is awaited, the current "context" is captured and used to resume the method when the Task completes. This "context" is the current `SynchronizationContext` unless it's null, in which case it's the current `TaskScheduler`. GUI and ASP.NET applications have a `SynchronizationContext` that permits only one chunk of code to run at a time. When the `await` completes, it attempts to execute the remainder of the async method within the captured context. But that context already has a thread in it, which is (synchronously) waiting for the async method to complete. They're each waiting for the other, causing a deadlock.

Note that console applications don't cause this deadlock. They have a thread pool `SynchronizationContext` instead of a one-chunk-at-a-time `SynchronizationContext`, so when the `await` completes, it schedules the remainder of the async method on a thread pool thread. The method is able to complete, which completes its returned task, and there's no deadlock. This difference in behavior can be confusing when programmers write a test console program, observe the partially async code work as expected, and then move the same code into a GUI or ASP.NET application, where it deadlocks.

The best solution to this problem is to allow async code to grow naturally through the codebase. If you follow this solution, you'll see async code expand to its entry point, usually an event handler or controller action. Console applications can't follow this solution fully because the `Main` method can't be async. If the `Main` method were async, it could return before it completed, causing the program to end. **Figure 4** demonstrates this exception to the guideline: The `Main` method for a console application is one of the few situations where code may block on an asynchronous method.

Figure 4 The Main Method May Call `Task.Wait` or `Task.Result`

XML	 Copy
<pre>class Program {     static void Main()     {         MainAsync().Wait();     }     static async Task MainAsync()     {         try         {             // Asynchronous implementation.             await Task.Delay(1000);         }         catch (Exception ex)</pre>	

```
{  
    // Handle exceptions.  
}  
}  
}
```

Allowing async to grow through the codebase is the best solution, but this means there's a lot of initial work for an application to see real benefit from async code. There are a few techniques for incrementally converting a large codebase to async code, but they're outside the scope of this article. In some cases, using `Task.Wait` or `Task.Result` can help with a partial conversion, but you need to be aware of the deadlock problem as well as the error-handling problem. I'll explain the error-handling problem now and show how to avoid the deadlock problem later in this article.

Every `Task` will store a list of exceptions. When you await a `Task`, the first exception is re-thrown, so you can catch the specific exception type (such as `InvalidOperationException`). However, when you synchronously block on a `Task` using `Task.Wait` or `Task.Result`, all of the exceptions are wrapped in an `AggregateException` and thrown. Refer again to **Figure 4**. The try/catch in `MainAsync` will catch a specific exception type, but if you put the try/catch in `Main`, then it will always catch an `AggregateException`. Error handling is much easier to deal with when you don't have an `AggregateException`, so I put the "global" try/catch in `MainAsync`.

So far, I've shown two problems with blocking on async code: possible deadlocks and more-complicated error handling. There's also a problem with using blocking code within an async method. Consider this simple example:

XML

 Copy

```
public static class NotFullyAsynchronousDemo  
{  
    // This method synchronously blocks a thread.  
    public static async Task TestNotFullyAsync()  
    {  
        await Task.Yield();  
        Thread.Sleep(5000);  
    }  
}
```

This method isn't fully asynchronous. It will immediately yield, returning an incomplete task, but when it resumes it will synchronously block whatever thread is running. If this method is called from a GUI context, it will block the GUI thread; if it's called from an ASP.NET request context, it will block the current ASP.NET request thread. Asynchronous

code works best if it doesn't synchronously block. **Figure 5** is a cheat sheet of async replacements for synchronous operations.

**Figure 5 The "Async Way" of Doing Things**


To Do This ...	Instead of This ...	Use This
Retrieve the result of a background task	Task.Wait or Task.Result	await
Wait for any task to complete	Task.WaitAny	await Task.WhenAny
Retrieve the results of multiple tasks	Task.WaitAll	await Task.WhenAll
Wait a period of time	Thread.Sleep	await Task.Delay

To summarize this second guideline, you should avoid mixing async and blocking code. Mixed async and blocking code can cause deadlocks, more-complex error handling and unexpected blocking of context threads. The exception to this guideline is the `Main` method for console applications, or—if you're an advanced user—managing a partially asynchronous codebase.

## Configure Context

Earlier in this article, I briefly explained how the "context" is captured by default when an incomplete `Task` is awaited, and that this captured context is used to resume the async method. The example in **Figure 3** shows how resuming on the context clashes with synchronous blocking to cause a deadlock. This context behavior can also cause another problem—one of performance. As asynchronous GUI applications grow larger, you might find many small parts of async methods all using the GUI thread as their context. This can cause sluggishness as responsiveness suffers from "thousands of paper cuts."

To mitigate this, await the result of `ConfigureAwait` whenever you can. The following code snippet illustrates the default context behavior and the use of `ConfigureAwait`:

XML	 Copy
<pre>async Task MyMethodAsync() {     // Code here runs in the original context.     await Task.Delay(1000);     // Code here runs in the original context.     await Task.Delay(1000).ConfigureAwait(</pre>	



```
    continueOnCapturedContext: false);  
    // Code here runs without the original  
    // context (in this case, on the thread pool).  
}
```

By using `ConfigureAwait`, you enable a small amount of parallelism: Some asynchronous code can run in parallel with the GUI thread instead of constantly badgering it with bits of work to do.

Aside from performance, `ConfigureAwait` has another important aspect: It can avoid deadlocks. Consider **Figure 3** again; if you add `"ConfigureAwait(false)"` to the line of code in `DelayAsync`, then the deadlock is avoided. This time, when the `await` completes, it attempts to execute the remainder of the async method within the thread pool context. The method is able to complete, which completes its returned task, and there's no deadlock. This technique is particularly useful if you need to gradually convert an application from synchronous to asynchronous.

If you can use `ConfigureAwait` at some point within a method, then I recommend you use it for every `await` in that method after that point. Recall that the context is captured only if an incomplete `Task` is awaited; if the `Task` is already complete, then the context isn't captured. Some tasks might complete faster than expected in different hardware and network situations, and you need to graciously handle a returned task that completes before it's awaited. **Figure 6** shows a modified example.

Figure 6 Handling a Returned Task that Completes Before It's Awaited


XML

 Copy

```
async Task MyMethodAsync()  
{  
    // Code here runs in the original context.  
    await Task.FromResult(1);  
    // Code here runs in the original context.  
    await Task.FromResult(1).ConfigureAwait(continueOnCapturedContext: false);  
    // Code here runs in the original context.  
    var random = new Random();  
    int delay = random.Next(2); // Delay is either 0 or 1  
    await Task.Delay(delay).ConfigureAwait(continueOnCapturedContext: false);  
    // Code here might or might not run in the original context.  
    // The same is true when you await any Task  
    // that might complete very quickly.  
}
```


You should not use `ConfigureAwait` when you have code after the `await` in the method that needs the context. For GUI apps, this includes any code that manipulates GUI elements, writes data-bound properties or depends on a GUI-specific type such as `Dispatcher/CoreDispatcher`. For ASP.NET apps, this includes any code that uses `HttpContext.Current` or builds an ASP.NET response, including return statements in controller actions. **Figure 7** demonstrates one common pattern in GUI apps—having an async event handler disable its control at the beginning of the method, perform some awaits and then re-enable its control at the end of the handler; the event handler can't give up its context because it needs to re-enable its control.

Figure 7 Having an Async Event Handler Disable and Re-Enable Its Control

XML	 Copy
<pre>private async void button1_Click(object sender, EventArgs e) {     button1.Enabled = false;     try     {         // Can't use ConfigureAwait here ...         await Task.Delay(1000);     }     finally     {         // Because we need the context here.         button1.Enabled = true;     } }</pre>	

Each async method has its own context, so if one async method calls another async method, their contexts are independent. **Figure 8** shows a minor modification of **Figure 7**.

Figure 8 Each Async Method Has Its Own Context

XML	 Copy
<pre>private async Task HandleClickAsync() {     // Can use ConfigureAwait here.     await Task.Delay(1000).ConfigureAwait(continueOnCapturedContext: false); } private async void button1_Click(object sender, EventArgs e) {     button1.Enabled = false;     try     {</pre>	

```
// Can't use ConfigureAwait here.
await HandleClickAsync();
}
finally
{
    // We are back on the original context for this method.
    button1.Enabled = true;
}
}
```

Context-free code is more reusable. Try to create a barrier in your code between the context-sensitive code and context-free code, and minimize the context-sensitive code. In **Figure 8**, I recommend putting all the core logic of the event handler within a testable and context-free async Task method, leaving only the minimal code in the context-sensitive event handler. Even if you’re writing an ASP.NET application, if you have a core library that’s potentially shared with desktop applications, consider using ConfigureAwait in the library code.

To summarize this third guideline, you should use ConfigureAwait when possible. Context-free code has better performance for GUI applications and is a useful technique for avoiding deadlocks when working with a partially async codebase. The exceptions to this guideline are methods that require the context.

# Know Your Tools

There’s a lot to learn about async and await, and it’s natural to get a little disoriented. **Figure 9** is a quick reference of solutions to common problems.

**Figure 9 Solutions to Common Async Problems**


Problem	Solution
Create a task to execute code	Task.Run or TaskFactory.StartNew ( <i>not</i> the Task constructor or Task.Start)
Create a task wrapper for an operation or event	TaskFactory.FromAsync or TaskCompletionSource<T>
Support cancellation	CancellationTokenSource and CancellationToken
Report progress	IProgress<T> and Progress<T>
Handle streams of data	TPL Dataflow or Reactive Extensions

Synchronize access to a shared resource	SemaphoreSlim
Asynchronously initialize a resource	AsyncLazy<T>
Async-ready producer/consumer structures	TPL Dataflow or AsyncCollection<T>

The first problem is task creation. Obviously, an async method can create a task, and that's the easiest option. If you need to run code on the thread pool, use `Task.Run`. If you want to create a task wrapper for an existing asynchronous operation or event, use `TaskCompletionSource<T>`. The next common problem is how to handle cancellation and progress reporting. The base class library (BCL) includes types specifically intended to solve these issues: `CancellationTokenSource/CancellationToken` and `IProgress<T>/Progress<T>`. Asynchronous code should use the Task-based Asynchronous Pattern, or TAP ([msdn.microsoft.com/library/55873175](https://msdn.microsoft.com/library/55873175)), which explains task creation, cancellation and progress reporting in detail.

Another problem that comes up is how to handle streams of asynchronous data. Tasks are great, but they can only return one object and only complete once. For asynchronous streams, you can use either TPL Dataflow or Reactive Extensions (Rx). TPL Dataflow creates a "mesh" that has an actor-like feel to it. Rx is more powerful and efficient but has a more difficult learning curve. Both TPL Dataflow and Rx have async-ready methods and work well with asynchronous code.

Just because your code is asynchronous doesn't mean that it's safe. Shared resources still need to be protected, and this is complicated by the fact that you can't await from inside a lock. Here's an example of async code that can corrupt shared state if it executes twice, even if it always runs on the same thread:

XML	 Copy
<pre>int value;  Task&lt;int&gt; GetNextValueAsync(int current);  async Task UpdateValueAsync() {     value = await GetNextValueAsync(value); }</pre>	

```
}
```

The problem is that the method reads the value and suspends itself at the `await`, and when the method resumes it assumes the value hasn't changed. To solve this problem, the `SemaphoreSlim` class was augmented with the async-ready `WaitAsync` overloads. **Figure 10** demonstrates `SemaphoreSlim.WaitAsync`.

Figure 10 SemaphoreSlim Permits Asynchronous Synchronization

XML

 Copy

```
SemaphoreSlim mutex = new SemaphoreSlim(1);

int value;

Task<int> GetNextValueAsync(int current);

async Task UpdateValueAsync()
{
    await mutex.WaitAsync().ConfigureAwait(false);

    try
    {
        value = await GetNextValueAsync(value);
    }

    finally
    {
        mutex.Release();
    }
}
```

Asynchronous code is often used to initialize a resource that's then cached and shared. There isn't a built-in type for this, but Stephen Toub developed an `AsyncLazy<T>` that acts like a merge of `Task<T>` and `Lazy<T>`. The original type is described on his blog

([bit.ly/dEN178](http://bit.ly/dEN178)), and an updated version is available in my AsyncEx library ([nitoasyncex.codeplex.com](http://nitoasyncex.codeplex.com)).

Finally, some async-ready data structures are sometimes needed. TPL Dataflow provides a `BufferBlock<T>` that acts like an async-ready producer/consumer queue. Alternatively, AsyncEx provides `AsyncCollection<T>`, which is an async version of `BlockingCollection<T>`.

I hope the guidelines and pointers in this article have been helpful. Async is a truly awesome language feature, and now is a great time to start using it!

---

**Stephen Cleary** *is a husband, father and programmer living in northern Michigan. He has worked with multithreading and asynchronous programming for 16 years and has used async support in the Microsoft .NET Framework since the first CTP. His home page, including his blog, is at [stephencleary.com](http://stephencleary.com).*

Thanks to the following technical expert for reviewing this article: Stephen Toub  
Stephen Toub works on the Visual Studio team at Microsoft. He specializes in areas related to parallelism and asynchrony.