# BEST PRACTICES FOR C# ASYNC/AWAIT

HTTP://CODE.JONWAGNER.COM/2012/09/06/BEST-PRACTICES-FOR-C-ASYNCAWAIT/

The async/await keywords make asynchronous programming much easier for developers to digest and implement correctly. They also hide a little bit of complexity that can sneak up and (bug) bite you if you aren't paying attention. It's worth reading and applying these best practices when doing .NET asynchronous programming.

A lot of this information is also in other places, but since it is so new, it bears repeating and explaining differently.

## General: Only use async for things that can take a "long" time

There is some overhead around creating **Tasks** and other structures to manage the asynchronous state. If your operation is long, such as making an I/O call, then the overhead will generally be negligible. However, if your operation is short, or might only consume CPU cycles, then it may be better to perform your operation synchronously.

In general, the .NET Framework 4.5 team has done a good job choosing which methods should be async, so if the Framework method ends in *Async* and returns a *task*, then you should probably use it asynchronously.

## General: Favor async/await to Tasks

In general, writing asynchronous code using **async/await** is much easier to code and to read than using Tasks.

```
public Task<Data> GetDataAsync()
{
    return MyWebService.FetchDataAsync()
        .ContinueWith(t => new Data (t.Result));
}

public async Task<Data> GetDataAsync()
{
    var result = await MyWebService.FetchDataAsync();

    return new Data (result);
}
```

In terms of performance, both methods have a little overhead, but they scale a little differently.

- Tasks build a continuation chain that increases according to the number of tasks chained together, and the state of the system is managed through closures captured by the compiler.

- Async/await builds a state machine that does not take more resources as you add steps, but the compiler may capture more state variables and state machine stack data depending on your code (and the compiler). See MSDN: "Async Performance: Understanding the Costs of Async and Await" for some great details.

In general, async/await should take fewer resources and run faster than Tasks in most scenarios.

## General: Use a static Empty Task for Conditionals

Sometimes you only want to spawn off a task if some condition is true. Unfortunately, await will throw a NullReferenceException on a null task, so empty tasks can make your code hard to read.

```
public async Task<Data> GetDataAsync(bool getLatestData)
{
    Task<WebData> task = null;
    if (getLatestData)
        task = MyWebService.FetchDataAsync();

    // do other stuff here

    // don't forget to check for null!
    WebData result = null;
    if (task != null)
        result = await task;

    return new Data (result);
}
```

One way to make this a little easier is to have an empty task that is already complete. Much simpler code:

```
public async Task<Data> GetDataAsync(bool getLatestData)
{
    var task = getLatestData ?
           MyWebService.FetchDataAsync() : Empty<WebData>.Task;

    // do other stuff here

    // task is always available
    return new Data (await task);
}
```

Make sure that the task is a static task, so it already starts out as completed. For example:

```
public static class Empty<T>
{
    public static Task<T> Task { get { return _task; } }

   private static readonly Task<T> _task =
          System.Threading.Tasks.Task.FromResult(default(T));
}
```

## Performance: Cache tasks rather than data if there is a performance issue

There is some overhead in creating tasks. If you are caching your results, but then converting them back to tasks, you may be creating extra instances of Tasks.

```
public Task<byte[]> GetContentsOfUrl(string url)
{
    byte[] bytes;

    if (_cache.TryGetValue(url, out bytes))
        // extra task created here:
        return Task<byte[]>.Factory.StartNew(() => bytes);

    bytes = MyWebService.GetContentsAsync(url)
        .ContinueWith(t => { _cache.Add(url, t.Result); return t.Result; );
}

// NOTE: this is not threadsafe (do not copy this code)
private static Dictionary<string, byte[]> _cache = new Dictionary<string, byte[]>();
```

If may be better to cache the Tasks themselves. Then the dependent code can wait on an already-completed Task. There are optimizations in the TPL and await code to take a “fast path” when waiting on a completed Task.

```
public Task<byte[]> GetContentsOfUrl(string url)
{
    Task<byte[]> bytes;

    if (!_cache.TryGetValue(url, out bytes))
    {
        bytes = MyWebService.GetContentsAsync(url);
        _cache.Add(url, bytes);
    }

    return bytes;
}

// NOTE: this is not threadsafe (do not copy this code)
private static Dictionary<string, Task<byte[]>> _cache = new Dictionary<string, Task<byte[]>>
```

## Performance: Understand await state

When you use async/await, the compiler creates a state machine that has some captured variables and a stack. For example:

```
public static async Task FooAsync()
{
  var data = await MyWebService.GetDataAsync();
  var otherData = await MyWebService.GetOtherDataAsync();
  Console.WriteLine(&quot;{0} = &quot;1&quot;, data, otherdata);
}
```

This will create a state object with a few variables. Note that the compiler will generally capture the variables in the method:

```
[StructLayout(LayoutKind.Sequential), CompilerGenerated]
private struct <FooAsync>d__0 : <>t__IStateMachine {
  private int <>1__state;
  public AsyncTaskMethodBuilder <>t__builder;
  public Action <>t__MoveNextDelegate;

  public Data <data>5__1;
  public OtherData <otherData>5__2;
  private object <>t__stack;
  private object <>t__awaiter;

  public void MoveNext();
  [DebuggerHidden]
  public void <>t__SetMoveNextDelegate(Action param0);
}
```

Note #1: so if you declare a variable, it will get captured in the state above. This may keep objects around longer than you expect.

Note #2: but if you don't declare a variable, and a value is used across awaits, the variable may be put on an internal stack:

```
public static async Task FooAsync()
{
  var data = MyWebService.GetDataAsync();
  var otherData = MyWebService.GetOtherDataAsync();

  // these interim results are spilled onto an internal stack
  // and there are extra context switches between the awaits
  Console.WriteLine("{0} = {1}", await data, await otherdata);
}
```

You shouldn't need to worry too much about this unless you are seeing performance problems. If you are worried about performance, MSDN has a great article: Async Performance: Understanding the Costs of Async and Await

## Stability: async/await is NOT Task.Wait

See my post on this. The state machine generated by async/await is not the same as Task.ContinueWith/Wait. In general, you can replace your Task logic with await, but there are some performance and stability issues that may come up. Keep reading for some guidance on this.

## Stability: Know your synchronization contexts

.NET code always executes in a context. The context defines the current user/security principal, and other things required by the framework. In some execution contexts, the code is in a synchronization context that manages the scheduling of the Tasks and other asynchronous work.

By default, await will continue the work in the context in which it was started. This is handy, because generally you will want the security context restored, and you want your code to have access to the Windows UI objects if it already had access to it.

NOTE: Task.Factory.StartNew does not maintain the context.

However, some synchronization contexts are non-reentrant and single-threaded. This means only one unit of work can be executed in the context at a given time. An example of this is the Windows UI thread or the ASP.NET request context.

In these single-threaded synchronization contexts, it's easy to deadlock yourself. If you spawn off a task from a single-threaded context, then wait for that task in the context, your waiting code may be blocking the background task.

```
public ActionResult ActionAsync()
{
    // DEADLOCK: this blocks on the async task
    // which is waiting to continue on this sync context
    var data = GetDataAsync().Result;

    return View(data);
}

private async Task<string> GetDataAsync()
{
    // a very simple async method
    var result = await MyWebService.GetDataAsync();
    return result.ToString();
}
```

## Stability: Don't Wait for Tasks Here

As a general rule, if you are writing asynchronous code, be careful when using **Wait**. (**await** is ok.)
Don't **Wait** for tasks in single-threaded synchronization contexts such as:

- UI Threads
- ASP.NET Contexts

The good news is that these libraries all allow you to return Tasks and the framework will wait for the completion for you. Let the framework do the waiting.

```
public async Task<ActionResult> ActionAsync()
{
    // this method uses async/await and returns a task
    var data = await GetDataAsync();

    return View(data);
}
```

If you are writing async libraries, your callers will have to write async code. This used to be a problem, as writing async code was tedious and error prone, but with async/await, a lot of the complexity is now handled by the compiler. Your code will have more reliable performance, and you are less likely to be blocked by the Evil .NET ThreadPool (yeah, I'm talking about *you*, ThreadPool).

## Stability: Consider ConfigureAwait for Library Tasks

If you **must** wait for a task in one of these contexts, you can use **ConfigureAwait** to tell the system that it does not need to run the background task in your context. The downside is that the background task will not have access to the same synchronization context, so you would lose access to the Windows UI or the HttpContext. (Your security context would still be captured.)
If you are writing a "library" function that generates a task, you may not know where it's going to be called from. So it might be safer to add ConfigureAwait(false) to your task before returning it.

```
private async Task<string> GetDataAsync()
{
    // ConfigureAwait(false) tells the system to
    // allow the remainder to run in any sync context
    var result = await MyWebService.GetDataAsync().ConfigureAwait(false);
    return result.ToString();
}
```

## Stability: Understand Exception behavior

When looking at async code, sometimes it is hard to tell what happens with exceptions. Will it be thrown to the caller or to the awaiter of the task?
The rules are straightforward, but sometimes it is hard to tell just by looking.

Some examples:

Exceptions thrown from **within an async/await method** will be sent to the **awaiter of the task**.

```
public async Task<Data> GetContentsOfUrl(string url)
{
    // this will be thrown to the awaiter of the task
    if (url == null) throw new ArgumentNullException();

    var data = await MyWebService.GetContentsOfUrl();
    return data.DoStuffToIt();
}
```

Exceptions thrown from **within a Task's action** will be sent to the**awaiter of the task**.

```
public Task<Data> GetContentsOfUrl(string url)
{
    return Task<Data>.Factory.StartNew(() =>
    {
        // this will be thrown to the awaiter of the task
        if (url == null) throw new ArgumentNullException();

        var data = await MyWebService.GetContentsOfUrl();
        return data.DoStuffToIt();
    }
}
```

Exceptions thrown while **setting up** a Task chain will be sent to the **caller of the method**.

```
public Task<Data> GetContentsOfUrl(string url)
{
    // this will be thrown to the caller of the method
    if (url == null) throw new ArgumentNullException();

    return Task<Data>.Factory.StartNew(() =>
    {
         var data = await MyWebService.GetContentsOfUrl();
        return data.DoStuffToIt();
    }
}
```

This last example is one of the reasons I prefer async/await to setting up Task chains. There are fewer places to handle exceptions.

## Additional Links

- MSDN Async/Await FAQ
- About the await "fast path"for performance
- MSDN: "Await, and UI, and deadlocks! Oh my!"
- MSDN: "Async Performance: Understanding the Costs of Async and Await"