Using Task.Run in Conjunction with Async/Await


pluralsight.com/guides/using-task-run-async-await

Async/Await on a Separate Thread

The `async/await` approach in C# is great in part because it isolates the asynchronous concept of *waiting* from other details. So when you `await` a predefined method in a third-party library or in .NET itself, you don't necessarily have to concern yourself with the nature of the operation you're awaiting. If a predefined method returns a `Task`, you simply mark the calling method as `async` and put the `await` keyword in front of the method call. It's helpful to understand the control flow associated with the await keyword, but that's basically it.

But what about when you need to call some predefined method that does not return a `Task`? If it is some trivial operation that executes quickly, then you can just call it synchronously, without the need for `await`. But if it is a long-running operation, you may need to find a way to make it asynchronous. This is especially so for applications that have a graphical user interface, which may appear frozen when performing a long-running operation synchronously.

One way to turn a synchronous operation into an asynchronous one is to run it on a separate thread, and that's where `Task.Run` comes in. The `Run` method queues code to run on a different thread (usually from the "thread pool", which is a set of worker threads managed for your application by .NET). And, importantly, `Task.Run` returns a `Task` which means you can use the `await` keyword with it!

So let's explore using `Task.Run` in conjunction with `async/await`. It's not difficult to use, but as we shall see, knowing *when* its use is appropriate is not so straightforward.

Task.Run at a Glance

Let's take a look at a simple example of `Task.Run`, to get an idea of the syntax:

```
1async void OnButtonClick()
2{
3  await Task.Run(() => /* your code here*/);
4}
```

csharp

`Task.Run` accepts an `Action` (or a `Func<T>` in the event you need to return a value), so it is very flexible. You can write your code in line, e.g.:

```
1await Task.Run(() => DoExpensiveOperation(someParameter));
```

csharp

...or within a block, e.g.:

```
await Task.Run(() =>
{
  for (int i = 0; i < int.MaxValue; i++)
  {
    ...
  }
});
```

csharp

In the case of a single method with no parameters, you simply pass the name of the method:

```
await Task.Run(MyMethod);
```

csharp

Regardless of the syntax used, execution happens in the same manner: The current thread is released and the code passed in is executed on a thread from the thread pool. When execution is complete, however, in which thread does the rest of the calling method execute? Most of the time you'll want it to continue on the original thread you were on when you called `await`, especially in the case of an application with a graphical user interface, where you'll need to update UI elements on the main application thread. Fortunately, `await` captures the current `SynchronizationContext`, which includes information about the current thread, and by default automatically returns to that thread when finished.

> Note: The behavior *without* `await` is trickier, so unless you understand those nuances be sure to use `await` with `Task.Run` to avoid unexpected behavior.

CPU vs. I/O-Bound Code

Let's tackle the question of *when* to use `Task.Run`. In order to do so, we need to understand the difference between CPU-bound and I/O-bound code. First, what is CPU-bound code? By saying something is "bound" by the CPU, we're basically saying that the computer's processor (or a particular thread running in the processor) is the bottleneck. Your processor is working as fast as it can to perform some calculation, but it is still taking long enough to cause a noticeable delay. In other words, the CPU is what's holding you back from faster performance in that code. You could also say that the delay is *caused* by the CPU.

By contrast, for I/O-bound code the data transfer rate of an input or output process is the bottleneck. This could be a local input/output process, such as reading or saving a file to local storage, or it could be communicating with a remote server to upload or download something. In both cases, the CPU is somewhat idle because it is waiting to finish sending or receiving some data. In the case of an unresponsive web address, the CPU would be almost completely idle, simply waiting until it can begin the data transfer.

As you can see, although in both cases there is a delay, the operations are quite different in nature. As you write code, try categorizing each asynchronous operation when you use `await`. Ask yourself if it is a CPU-bound operation or an I/O-bound one. But does that really matter?

Know the Nature of Your Asynchronous Operation

Yes, the nature of your asynchronous operation matters. The short explanation is that **launching an operation on a separate thread via `Task.Run` is mainly useful for CPU-bound operations, not I/O-bound operations.** But why is that?

Most modern computer processors have multiple cores, which is a bit like having multiple processors. When your application begins running, the .NET runtime creates a pool of threads for your application's use. The default size of this pool often corresponds to the number of cores in your processor. If the main thread of your application is keeping a core busy, you can take advantage of the other cores by passing off some work using `Task.Run`.

> Note that although related, thread ≠ core. A thread is a software concept and a core is a hardware component. You can create any number of threads, but the number of cores is fixed. That said, when you call `Task.Run` the thread will run on a different core if available because that's what will provide the greatest performance advantages.

So that's `Task.Run` with a CPU-bound operation. If you use `Task.Run` with an I/O operation, you're creating a thread (and probably occupying a CPU core) that will mostly be waiting. It may be a quick and easy way to keep your application responsive, but it's not the most efficient use of system resources. A much better approach is to use `await` *without* `Task.Run` for I/O operations. This will have the same positive effect of keeping your application responsive, but without wastefully occupying additional threads/cores.

> In order to use `await` without `Task.Run` for I/O operations, you'll need to use asynchronous methods that return `Task` without resorting to calling `Task.Run` itself. This is straightforward when working with certain classes built into .NET such as `FileStream` and `HttpClient`, which provide asynchronous methods for that exact purpose. But you may find that some classes in .NET or in third-party libraries only provide synchronous methods, in which case you may be forced to use `Task.Run` to achieve asynchrony even though it is just an I/O operation. Also, although it is not a recommended practice, there are some third-party libraries that simply "wrap" calls to synchronous methods with a call to `Task.Run`, essentially invoking `Task.Run` on your behalf. It may not cause any problems for your particular application, but be aware of those possibilities, and prefer asynchronous I/O operations without `Task.Run`.

Downloading and Blurring an Image

Let's reinforce the above concepts by considering a more specific example. In previous guides in this series, we've written an application that downloads an image from the internet and saves a blurred version of that image. We broke down the necessary

operations into three methods:

```
1static Task<byte[]> DownloadImage(string url) { ... }
2
3static Task<byte[]> BlurImage(string imagePath) { ... }
4
5static Task SaveImage(byte[] bytes, string imagePath) { ... }
```

csharp

It's time to fully implement these methods, keeping in mind what we've just learned. Clearly, downloading an image and saving an image to disk are both I/O operations, so let's try *not* to use `Task.Run` for those. This is easily done using .NET's `HttpClient` and `FileStream` methods ending with "Async".

```
1static Task<byte[]> DownloadImage(string url)
2{
3  var client = new HttpClient();
4  return client.GetByteArrayAsync(url);
5}
6
7static async Task SaveImage(byte[] bytes, string imagePath)
8{
9  using (var fileStream = new FileStream(imagePath, FileMode.Create))
10  {
11    await fileStream.WriteAsync(bytes, 0, bytes.Length);
12  }
13}
```

csharp

By contrast, *blurring* an image (like any image, video, or audio processing) is very CPU intensive because it has to make many calculations to determine what the resulting image will look like. To understand this better, an even simpler example would be darkening an image. A naive implementation of an image darkener would subtract a constant value from the Red, Green, and Blue values in the color of each pixel in the image, bringing the values closer to zero. Image processing boils down to arithmetic and arithmetic happens *in the CPU*. So to blur the image, we'll do so on a separate thread using `Task.Run`. In this example, I'm using a library called ImageSharp. It's available as `SixLabors.ImageSharp` in NuGet; I'm using version `1.0.0-beta0006`.

```
1static async Task<byte[]> BlurImage(string imagePath)
2{
3  return await Task.Run(() =>
4  {
5    var image = Image.Load(imagePath);
6    image.Mutate(ctx => ctx.GaussianBlur());
7    using (var memoryStream = new MemoryStream())
8    {
9      image.SaveAsJpeg(memoryStream);
10      return memoryStream.ToArray();
11    }
12  });
13}
```

csharp

The main reason we need `Task.Run` here is the call to `image.Mutate`, although the call to `image.SaveAsJpeg` is also CPU intensive because it is compressing the image. We are, however, also performing a bit of I/O at the beginning with the call to `Image.Load`. There is no asynchronous version of `Image.Load` available in the ImageSharp library, so we do it together with everything else as part of the call to `Task.Run`.

As you can see, sometimes you'll have a mixture of CPU-bound and I/O-bound operations in a single method. Since `Image.Load` also accepts a byte array, one improvement we could make would be to add a fourth method called `LoadImage` where we read the file into a byte array using `FileStream`, and then in `BlurImage` we could accept a byte array instead of the image path. This would get us a bit closer to having a blur method that is purely CPU-bound. You'll have to use your best judgment when structuring your code, both in terms of readability as well as system resource efficiency.

Conclusion

Once you understand what `Task.Run` does and when it is appropriate, it's an incredibly useful tool to have at your fingertips, especially when used in conjunction with `async/await`. The knowledge doesn't stop there, however! In the next guide in this series, we'll look at a few advanced tips for using `Task.Run`, to make your life easier as your application grows in complexity.