

# Consuming the Task-based Asynchronous Pattern

03/30/2017 • 23 minutes to read •  +11

## In this article

[Suspending Execution with Await](#)

[Canceling an Asynchronous Operation](#)

[Monitoring Progress](#)

[Using the Built-in Task-based Combinators](#)

[Building Task-based Combinators](#)

[Building Task-based Data Structures](#)

[See also](#)

When you use the Task-based Asynchronous Pattern (TAP) to work with asynchronous operations, you can use callbacks to achieve waiting without blocking. For tasks, this is achieved through methods such as [Task.ContinueWith](#). Language-based asynchronous support hides callbacks by allowing asynchronous operations to be awaited within normal control flow, and compiler-generated code provides this same API-level support.

## Suspending Execution with Await

Starting with the .NET Framework 4.5, you can use the [await](#) keyword in C# and the [Await Operator](#) in Visual Basic to asynchronously await [Task](#) and [Task<TResult>](#) objects. When you're awaiting a [Task](#), the `await` expression is of type `void`. When you're awaiting a [Task<TResult>](#), the `await` expression is of type `TResult`. An `await` expression must occur inside the body of an asynchronous method. For more information about C# and Visual Basic language support in the .NET Framework 4.5, see the C# and Visual Basic language specifications.

Under the covers, the `await` functionality installs a callback on the task by using a continuation. This callback resumes the asynchronous method at the point of suspension. When the asynchronous method is resumed, if the awaited operation completed successfully and was a [Task<TResult>](#), its `TResult` is returned. If the [Task](#) or [Task<TResult>](#) that was awaited ended in the [Canceled](#) state, an [OperationCanceledException](#) exception is thrown. If the [Task](#) or [Task<TResult>](#) that was awaited ended in the [Faulted](#) state, the

exception that caused it to fault is thrown. A `Task` can fault as a result of multiple exceptions, but only one of these exceptions is propagated. However, the [Task.Exception](#) property returns an [AggregateException](#) exception that contains all the errors.

If a synchronization context ([SynchronizationContext](#) object) is associated with the thread that was executing the asynchronous method at the time of suspension (for example, if the [SynchronizationContext.Current](#) property is not `null`), the asynchronous method resumes on that same synchronization context by using the context's [Post](#) method. Otherwise, it relies on the task scheduler ([TaskScheduler](#) object) that was current at the time of suspension. Typically, this is the default task scheduler ([TaskScheduler.Default](#)), which targets the thread pool. This task scheduler determines whether the awaited asynchronous operation should resume where it completed or whether the resumption should be scheduled. The default scheduler typically allows the continuation to run on the thread that the awaited operation completed.

When an asynchronous method is called, it synchronously executes the body of the function up until the first `await` expression on an awaitable instance that has not yet completed, at which point the invocation returns to the caller. If the asynchronous method does not return `void`, a [Task](#) or [Task<TResult>](#) object is returned to represent the ongoing computation. In a non-void asynchronous method, if a return statement is encountered or the end of the method body is reached, the task is completed in the [RanToCompletion](#) final state. If an unhandled exception causes control to leave the body of the asynchronous method, the task ends in the [Faulted](#) state. If that exception is an [OperationCanceledException](#), the task instead ends in the [Canceled](#) state. In this manner, the result or exception is eventually published.

There are several important variations of this behavior. For performance reasons, if a task has already completed by the time the task is awaited, control is not yielded, and the function continues to execute. Additionally, returning to the original context isn't always the desired behavior and can be changed; this is described in more detail in the next section.

## Configuring Suspension and Resumption with `Yield` and `ConfigureAwait`


Several methods provide more control over an asynchronous method's execution. For example, you can use the [Task.Yield](#) method to introduce a yield point into the asynchronous method:

```
C#
```




```
public class Task : ...
{
    public static YieldAwaitable Yield();
    ...
}
```

This is equivalent to asynchronously posting or scheduling back to the current context.

C#	 Copy
<pre>Task.Run(async delegate {     for(int i=0; i&lt;1000000; i++)     {         await Task.Yield(); // fork the continuation into a separate work item         ...     } });</pre>	

You can also use the [Task.ConfigureAwait](#) method for better control over suspension and resumption in an asynchronous method. As mentioned previously, by default, the current context is captured at the time an asynchronous method is suspended, and that captured context is used to invoke the asynchronous method's continuation upon resumption. In many cases, this is the exact behavior you want. In other cases, you may not care about the continuation context, and you can achieve better performance by avoiding such posts back to the original context. To enable this, use the [Task.ConfigureAwait](#) method to inform the await operation not to capture and resume on the context, but to continue execution wherever the asynchronous operation that was being awaited completed:


C#	 Copy
<pre>await someTask.ConfigureAwait(continueOnCapturedContext:false);</pre>	

## Canceling an Asynchronous Operation


Starting with the .NET Framework 4, TAP methods that support cancellation provide at least one overload that accepts a cancellation token ([CancellationToken](#) object).

A cancellation token is created through a cancellation token source ([CancellationTokenSource](#) object). The source's [Token](#) property returns the cancellation token that will be signaled when the source's [Cancel](#) method is called. For example, if you


want to download a single webpage and you want to be able to cancel the operation, you create a [CancellationTokenSource](#) object, pass its token to the TAP method, and then call the source's [Cancel](#) method when you're ready to cancel the operation:

C#	 Copy
<pre>var cts = new CancellationTokenSource(); string result = await DownloadStringAsync(url, cts.Token); ... // at some point later, potentially on another thread cts.Cancel();</pre>	

To cancel multiple asynchronous invocations, you can pass the same token to all invocations:

C#	 Copy
<pre>var cts = new CancellationTokenSource(); IList&lt;string&gt; results = await Task.WhenAll(from url in urls select DownloadStringAsync(url, cts.Token)); // at some point later, potentially on another thread ... cts.Cancel();</pre>	

Or, you can pass the same token to a selective subset of operations:

C#	 Copy
<pre>var cts = new CancellationTokenSource(); byte [] data = await DownloadDataAsync(url, cts.Token); await SaveToDiskAsync(outputPath, data, CancellationToken.None); ... // at some point later, potentially on another thread cts.Cancel();</pre>	

Cancellation requests may be initiated from any thread.

You can pass the [CancellationToken.None](#) value to any method that accepts a cancellation token to indicate that cancellation will never be requested. This causes the [CancellationToken.CanBeCanceled](#) property to return `false`, and the called method can optimize accordingly. For testing purposes, you can also pass in a pre-canceled cancellation token that is instantiated by using the constructor that accepts a Boolean value to indicate whether the token should start in an already-canceled or not-cancelable state.

This approach to cancellation has several advantages:

- You can pass the same cancellation token to any number of asynchronous and synchronous operations.
- The same cancellation request may be proliferated to any number of listeners.
- The developer of the asynchronous API is in complete control of whether cancellation may be requested and when it may take effect.
- The code that consumes the API may selectively determine the asynchronous invocations that cancellation requests will be propagated to.

## Monitoring Progress

Some asynchronous methods expose progress through a progress interface passed into the asynchronous method. For example, consider a function which asynchronously downloads a string of text, and along the way raises progress updates that include the percentage of the download that has completed thus far. Such a method could be consumed in a Windows Presentation Foundation (WPF) application as follows:

C#

 Copy

```
private async void btnDownload_Click(object sender, RoutedEventArgs e)
{
    btnDownload.IsEnabled = false;
    try
    {
        txtResult.Text = await DownloadStringAsync(txtUrl.Text,
            new Progress<int>(p => pbDownloadProgress.Value = p));
    }
    finally { btnDownload.IsEnabled = true; }
}
```

## Using the Built-in Task-based Combinators

The [System.Threading.Tasks](https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/consuming-the-task-based-asynchronous-pattern) namespace includes several methods for composing and working with tasks.

### Task.Run

The [Task](#) class includes several [Run](#) methods that let you easily offload work as a [Task](#) or [Task<TResult>](#) to the thread pool, for example:

C#



```
public async void button1_Click(object sender, EventArgs e)
{
    textBox1.Text = await Task.Run(() =>
    {
        // ... do compute-bound work here
        return answer;
    });
}
```

Some of these [Run](#) methods, such as the [Task.Run\(Func<Task>\)](#) overload, exist as shorthand for the [TaskFactory.StartNew](#) method. Other overloads, such as [Task.Run\(Func<Task>\)](#), enable you to use `await` within the offloaded work, for example:

C#



```
public async void button1_Click(object sender, EventArgs e)
{
    pictureBox1.Image = await Task.Run(async () =>
    {
        using(Bitmap bmp1 = await DownloadFirstImageAsync())
        using(Bitmap bmp2 = await DownloadSecondImageAsync())
        return Mashup(bmp1, bmp2);
    });
}
```

Such overloads are logically equivalent to using the [TaskFactory.StartNew](#) method in conjunction with the [Unwrap](#) extension method in the Task Parallel Library.

## Task.FromResult

Use the [FromResult](#) method in scenarios where data may already be available and just needs to be returned from a task-returning method lifted into a [Task<TResult>](#):

C#



```
public Task<int> GetValueAsync(string key)
{
    int cachedValue;
    return TryGetCachedValue(out cachedValue) ?
        Task.FromResult(cachedValue) :
```

```
        GetValueAsyncInternal();  
    }  
  
    private async Task<int> GetValueAsyncInternal(string key)  
    {  
        ...  
    }
```

## Task.WhenAll

Use the [WhenAll](#) method to asynchronously wait on multiple asynchronous operations that are represented as tasks. The method has multiple overloads that support a set of non-generic tasks or a non-uniform set of generic tasks (for example, asynchronously waiting for multiple void-returning operations, or asynchronously waiting for multiple value-returning methods where each value may have a different type) and to support a uniform set of generic tasks (such as asynchronously waiting for multiple `TReturn`-returning methods).

Let's say you want to send email messages to several customers. You can overlap sending the messages so you're not waiting for one message to complete before sending the next. You can also find out when the send operations have completed and whether any errors have occurred:

C#

 Copy

```
IEnumerable<Task> asyncOps = from addr in addrs select SendMailAsync(addr);  
await Task.WhenAll(asyncOps);
```


This code doesn't explicitly handle exceptions that may occur, but lets exceptions propagate out of the `await` on the resulting task from [WhenAll](#). To handle the exceptions, you can use code such as the following:

C#

 Copy

```
IEnumerable<Task> asyncOps = from addr in addrs select SendMailAsync(addr);  
try  
{  
    await Task.WhenAll(asyncOps);  
}  
catch(Exception exc)  
{  
    ...  
}
```


In this case, if any asynchronous operation fails, all the exceptions will be consolidated in an [AggregateException](#) exception, which is stored in the [Task](#) that is returned from the [WhenAll](#) method. However, only one of those exceptions is propagated by the `await` keyword. If you want to examine all the exceptions, you can rewrite the previous code as follows:

C#	 Copy
<pre>Task [] asyncOps = (from addr in addrs select SendMailAsync(addr)).ToArray(); try {     await Task.WhenAll(asyncOps); } catch(Exception exc) {     foreach(Task faulted in asyncOps.Where(t =&gt; t.IsFaulted))     {         ... // work with faulted and faulted.Exception     } }</pre>	

Let's consider an example of downloading multiple files from the web asynchronously. In this case, all the asynchronous operations have homogeneous result types, and it's easy to access the results:

C#	 Copy
<pre>string [] pages = await Task.WhenAll(     from url in urls select DownloadStringAsync(url));</pre>	

You can use the same exception-handling techniques we discussed in the previous void-returning scenario:

C#	 Copy
<pre>Task [] asyncOps =     (from url in urls select DownloadStringAsync(url)).ToArray(); try {     string [] pages = await Task.WhenAll(asyncOps);     ... } catch(Exception exc) {     foreach(Task&lt;string&gt; faulted in asyncOps.Where(t =&gt; t.IsFaulted))</pre>	



```
{  
    ... // work with faulted and faulted.Exception  
}  
}
```

## Task.WhenAny

You can use the [WhenAny](#) method to asynchronously wait for just one of multiple asynchronous operations represented as tasks to complete. This method serves four primary use cases:

- Redundancy: Performing an operation multiple times and selecting the one that completes first (for example, contacting multiple stock quote web services that will produce a single result and selecting the one that completes the fastest).
- Interleaving: Launching multiple operations and waiting for all of them to complete, but processing them as they complete.
- Throttling: Allowing additional operations to begin as others complete. This is an extension of the interleaving scenario.
- Early bailout: For example, an operation represented by task t1 can be grouped in a [WhenAny](#) task with another task t2, and you can wait on the [WhenAny](#) task. Task t2 could represent a time-out, or cancellation, or some other signal that causes the [WhenAny](#) task to complete before t1 completes.

## Redundancy

Consider a case where you want to make a decision about whether to buy a stock. There are several stock recommendation web services that you trust, but depending on daily load, each service can end up being slow at different times. You can use the [WhenAny](#) method to receive a notification when any operation completes:

C#

 Copy

```
var recommendations = new List<Task<bool>>()  
{  
    GetBuyRecommendation1Async(symbol),  
    GetBuyRecommendation2Async(symbol),  
    GetBuyRecommendation3Async(symbol)  
};  
Task<bool> recommendation = await Task.WhenAny(recommendations);  
if (await recommendation) BuyStock(symbol);
```

Unlike [WhenAll](#), which returns the unwrapped results of all tasks that completed successfully, [WhenAny](#) returns the task that completed. If a task fails, it's important to know that it failed, and if a task succeeds, it's important to know which task the return value is associated with. Therefore, you need to access the result of the returned task, or further await it, as this example shows.

As with [WhenAll](#), you have to be able to accommodate exceptions. Because you receive the completed task back, you can await the returned task to have errors propagated, and try/catch them appropriately; for example:

C#

 Copy

```
Task<bool> [] recommendations = ...;
while(recommendations.Count > 0)
{
    Task<bool> recommendation = await Task.WhenAny(recommendations);
    try
    {
        if (await recommendation) BuyStock(symbol);
        break;
    }
    catch(WebException exc)
    {
        recommendations.Remove(recommendation);
    }
}
```

Additionally, even if a first task completes successfully, subsequent tasks may fail. At this point, you have several options for dealing with exceptions: You can wait until all the launched tasks have completed, in which case you can use the [WhenAll](#) method, or you can decide that all exceptions are important and must be logged. For this, you can use continuations to receive a notification when tasks have completed asynchronously:

C#

 Copy

```
foreach(Task recommendation in recommendations)
{
    var ignored = recommendation.ContinueWith(
        t => { if (t.IsFaulted) Log(t.Exception); });
}
```

or:

C#

 Copy

```
foreach(Task recommendation in recommendations)
{
    var ignored = recommendation.ContinueWith(
        t => Log(t.Exception), TaskContinuationOptions.OnlyOnFaulted);
}
```

or even:

C#

 Copy

```
private static async void LogCompletionIfFailed(IEnumerable<Task> tasks)
{
    foreach(var task in tasks)
    {
        try { await task; }
        catch(Exception exc) { Log(exc); }
    }
}
...
LogCompletionIfFailed(recommendations);
```

Finally, you may want to cancel all the remaining operations:

C#

 Copy


```
var cts = new CancellationTokenSource();
var recommendations = new List<Task<bool>>()
{
    GetBuyRecommendation1Async(symbol, cts.Token),
    GetBuyRecommendation2Async(symbol, cts.Token),
    GetBuyRecommendation3Async(symbol, cts.Token)
};

Task<bool> recommendation = await Task.WhenAny(recommendations);
cts.Cancel();
if (await recommendation) BuyStock(symbol);
```


## Interleaving

Consider a case where you're downloading images from the web and processing each image (for example, adding the image to a UI control). You have to do the processing sequentially on the UI thread, but you want to download the images as concurrently as

possible. Also, you don't want to hold up adding the images to the UI until they're all downloaded—you want to add them as they complete:

C#	 Copy
<pre>List&lt;Task&lt;Bitmap&gt;&gt; imageTasks =     (from imageUrl in urls select GetBitmapAsync(imageUrl)).ToList(); while(imageTasks.Count &gt; 0) {     try     {         Task&lt;Bitmap&gt; imageTask = await Task.WhenAny(imageTasks);         imageTasks.Remove(imageTask);          Bitmap image = await imageTask;         panel.AddImage(image);     }     catch{} }</pre>	

You can also apply interleaving to a scenario that involves computationally intensive processing on the [ThreadPool](#) of the downloaded images; for example:

C#	 Copy
<pre>List&lt;Task&lt;Bitmap&gt;&gt; imageTasks =     (from imageUrl in urls select GetBitmapAsync(imageUrl)         .ContinueWith(t =&gt; ConvertImage(t.Result)).ToList()); while(imageTasks.Count &gt; 0) {     try     {         Task&lt;Bitmap&gt; imageTask = await Task.WhenAny(imageTasks);         imageTasks.Remove(imageTask);          Bitmap image = await imageTask;         panel.AddImage(image);     }     catch{} }</pre>	

## Throttling

Consider the interleaving example, except that the user is downloading so many images that the downloads have to be throttled; for example, you want only a specific number of downloads to happen concurrently. To achieve this, you can start a subset of the

asynchronous operations. As operations complete, you can start additional operations to take their place:

C#

 Copy

```
const int CONCURRENCY_LEVEL = 15;
Uri [] urls = ...;
int nextIndex = 0;
var imageTasks = new List<Task<Bitmap>>();
while(nextIndex < CONCURRENCY_LEVEL && nextIndex < urls.Length)
{
    imageTasks.Add(GetBitmapAsync(urls[nextIndex]));
    nextIndex++;
}

while(imageTasks.Count > 0)
{
    try
    {
        Task<Bitmap> imageTask = await Task.WhenAny(imageTasks);
        imageTasks.Remove(imageTask);

        Bitmap image = await imageTask;
        panel.AddImage(image);
    }
    catch(Exception exc) { Log(exc); }

    if (nextIndex < urls.Length)
    {
        imageTasks.Add(GetBitmapAsync(urls[nextIndex]));
        nextIndex++;
    }
}
```

## Early Bailout

Consider that you're waiting asynchronously for an operation to complete while simultaneously responding to a user's cancellation request (for example, the user clicked a cancel button). The following code illustrates this scenario:

C#

 Copy

```
private CancellationTokensource m_cts;

public void btnCancel_Click(object sender, EventArgs e)
{
    if (m_cts != null) m_cts.Cancel();
}
```

```
}

public async void btnRun_Click(object sender, EventArgs e)
{
    m_cts = new CancellationTokensource();
    btnRun.Enabled = false;
    try
    {
        Task<Bitmap> imageDownload = GetBitmapAsync(txtUrl.Text);
        await UntilCompletionOrCancellation(imageDownload, m_cts.Token);
        if (imageDownload.IsCompleted)
        {
            Bitmap image = await imageDownload;
            panel.AddImage(image);
        }
        else imageDownload.ContinueWith(t => Log(t));
    }
    finally { btnRun.Enabled = true; }
}

private static async Task UntilCompletionOrCancellation(
    Task asyncOp, CancellationTokens ct)
{
    var tcs = new TaskCompletionSource<bool>();
    using(ct.Register(() => tcs.TrySetResult(true)))
        await Task.WhenAny(asyncOp, tcs.Task);
    return asyncOp;
}
```

This implementation re-enables the user interface as soon as you decide to bail out, but doesn't cancel the underlying asynchronous operations. Another alternative would be to cancel the pending operations when you decide to bail out, but not reestablish the user interface until the operations actually complete, potentially due to ending early due to the cancellation request:

C#



```
private CancellationTokensource m_cts;

public async void btnRun_Click(object sender, EventArgs e)
{
    m_cts = new CancellationTokensource();

    btnRun.Enabled = false;
    try
    {
        Task<Bitmap> imageDownload = GetBitmapAsync(txtUrl.Text, m_cts.Token);
        await UntilCompletionOrCancellation(imageDownload, m_cts.Token);
    }
}
```

```
        Bitmap image = await imageDownload;
        panel.AddImage(image);
    }
    catch (OperationCanceledException) {}
    finally { btnRun.Enabled = true; }
}
```

Another example of early bailout involves using the [WhenAny](#) method in conjunction with the [Delay](#) method, as discussed in the next section.

## Task.Delay

You can use the [Task.Delay](#) method to introduce pauses into an asynchronous method's execution. This is useful for many kinds of functionality, including building polling loops and delaying the handling of user input for a predetermined period of time. The [Task.Delay](#) method can also be useful in combination with [Task.WhenAny](#) for implementing time-outs on awaits.

If a task that's part of a larger asynchronous operation (for example, an ASP.NET web service) takes too long to complete, the overall operation could suffer, especially if it fails to ever complete. For this reason, it's important to be able to time out when waiting on an asynchronous operation. The synchronous [Task.Wait](#), [Task.WaitAll](#), and [Task.WaitAny](#) methods accept time-out values, but the corresponding [TaskFactory.ContinueWhenAll/Task.WhenAny](#) and the previously mentioned [Task.WhenAll/Task.WhenAny](#) methods do not. Instead, you can use [Task.Delay](#) and [Task.WhenAny](#) in combination to implement a time-out.

For example, in your UI application, let's say that you want to download an image and disable the UI while the image is downloading. However, if the download takes too long, you want to re-enable the UI and discard the download:

C#



```
public async void btnDownload_Click(object sender, EventArgs e)
{
    btnDownload.Enabled = false;
    try
    {
        Task<Bitmap> download = GetBitmapAsync(url);
        if (download == await Task.WhenAny(download, Task.Delay(3000)))
        {
            Bitmap bmp = await download;
            pictureBox.Image = bmp;
            status.Text = "Downloaded";
        }
    }
}
```

```
    }
    else
    {
        pictureBox.Image = null;
        status.Text = "Timed out";
        var ignored = download.ContinueWith(
            t => Trace("Task finally completed"));
    }
}
finally { btnDownload.Enabled = true; }
```

The same applies to multiple downloads, because [WhenAll](#) returns a task:

C#



```
public async void btnDownload_Click(object sender, RoutedEventArgs e)
{
    btnDownload.Enabled = false;
    try
    {
        Task<Bitmap[]> downloads =
            Task.WhenAll(from url in urls select GetBitmapAsync(url));
        if (downloads == await Task.WhenAny(downloads, Task.Delay(3000)))
        {
            foreach(var bmp in downloads) panel.AddImage(bmp);
            status.Text = "Downloaded";
        }
        else
        {
            status.Text = "Timed out";
            downloads.ContinueWith(t => Log(t));
        }
    }
    finally { btnDownload.Enabled = true; }
```

## Building Task-based Combinators

Because a task is able to completely represent an asynchronous operation and provide synchronous and asynchronous capabilities for joining with the operation, retrieving its results, and so on, you can build useful libraries of combinators that compose tasks to build larger patterns. As discussed in the previous section, the .NET Framework includes several built-in combinators, but you can also build your own. The following sections provide several examples of potential combinator methods and types.



## RetryOnFault

In many situations, you may want to retry an operation if a previous attempt fails. For synchronous code, you might build a helper method such as `RetryOnFault` in the following example to accomplish this:

C#

 Copy

```
public static T RetryOnFault<T>(
    Func<T> function, int maxTries)
{
    for(int i=0; i<maxTries; i++)
    {
        try { return function(); }
        catch { if (i == maxTries-1) throw; }
    }
    return default(T);
}
```

You can build an almost identical helper method for asynchronous operations that are implemented with TAP and thus return tasks:

C#

 Copy

```
public static async Task<T> RetryOnFault<T>(
    Func<Task<T>> function, int maxTries)
{
    for(int i=0; i<maxTries; i++)
    {
        try { return await function().ConfigureAwait(false); }
        catch { if (i == maxTries-1) throw; }
    }
    return default(T);
}
```

You can then use this combinator to encode retries into the application's logic; for example:

C#

 Copy

```
// Download the URL, trying up to three times in case of failure
string pageContents = await RetryOnFault(
    () => DownloadStringAsync(url), 3);
```

You could extend the `RetryOnFault` function further. For example, the function could accept another `Func<Task>` that will be invoked between retries to determine when to try the operation again; for example:

C#

 Copy

```
public static async Task<T> RetryOnFault<T>(
    Func<Task<T>> function, int maxTries, Func<Task> retryWhen)
{
    for(int i=0; i<maxTries; i++)
    {
        try { return await function().ConfigureAwait(false); }
        catch { if (i == maxTries-1) throw; }
        await retryWhen().ConfigureAwait(false);
    }
    return default(T);
}
```

You could then use the function as follows to wait for a second before retrying the operation:

C#

 Copy

```
// Download the URL, trying up to three times in case of failure,
// and delaying for a second between retries
string pageContents = await RetryOnFault(
    () => DownloadStringAsync(url), 3, () => Task.Delay(1000));
```

## NeedOnlyOne

Sometimes, you can take advantage of redundancy to improve an operation's latency and chances for success. Consider multiple web services that provide stock quotes, but at various times of the day, each service may provide different levels of quality and response times. To deal with these fluctuations, you may issue requests to all the web services, and as soon as you get a response from one, cancel the remaining requests. You can implement a helper function to make it easier to implement this common pattern of launching multiple operations, waiting for any, and then canceling the rest. The `NeedOnlyOne` function in the following example illustrates this scenario:

C#

 Copy

```
public static async Task<T> NeedOnlyOne(
    params Func<CancellationToken, Task<T>> [] functions)
```

```
{
    var cts = new CancellationTokenSource();
    var tasks = (from function in functions
                  select function(cts.Token)).ToArray();
    var completed = await Task.WhenAny(tasks).ConfigureAwait(false);
    cts.Cancel();
    foreach(var task in tasks)
    {
        var ignored = task.ContinueWith(
            t => Log(t), TaskContinuationOptions.OnlyOnFaulted);
    }
    return completed;
}
```

You can then use this function as follows:

C#

 Copy

```
double currentPrice = await NeedOnlyOne(
    ct => GetCurrentPriceFromServer1Async("msft", ct),
    ct => GetCurrentPriceFromServer2Async("msft", ct),
    ct => GetCurrentPriceFromServer3Async("msft", ct));
```

## Interleaved Operations

There is a potential performance problem with using the [WhenAny](#) method to support an interleaving scenario when you're working with very large sets of tasks. Every call to [WhenAny](#) results in a continuation being registered with each task. For N number of tasks, this results in  $O(N^2)$  continuations created over the lifetime of the interleaving operation. If you're working with a large set of tasks, you can use a combinator (Interleaved in the following example) to address the performance issue:

C#

 Copy

```
static IEnumerable<Task<T>> Interleaved<T>(IEnumerable<Task<T>> tasks)
{
    var inputTasks = tasks.ToList();
    var sources = (from _ in Enumerable.Range(0, inputTasks.Count)
                   select new TaskCompletionSource<T>()).ToList();
    int nextTaskIndex = -1;
    foreach (var inputTask in inputTasks)
    {
        inputTask.ContinueWith(completed =>
        {
            var source = sources[Interlocked.Increment(ref nextTaskIndex)];
            if (completed.IsFaulted)
```

```

        source.TrySetException(completed.Exception.InnerExceptions);
    else if (completed.IsCanceled)
        source.TrySetCanceled();
    else
        source.TrySetResult(completed.Result);
}, CancellationToken.None,
TaskContinuationOptions.ExecuteSynchronously,
TaskScheduler.Default);
}
return from source in sources
select source.Task;
}

```

You can then use the combinator to process the results of tasks as they complete; for example:

C#

 Copy

```

IEnumerable<Task<int>> tasks = ...;
foreach(var task in Interleaved(tasks))
{
    int result = await task;
    ...
}

```

## WhenAllOrFirstException

In certain scatter/gather scenarios, you might want to wait for all tasks in a set, unless one of them faults, in which case you want to stop waiting as soon as the exception occurs. You can accomplish that with a combinator method such as `WhenAllOrFirstException` in the following example:

C#

 Copy

```

public static Task<T[]> WhenAllOrFirstException<T>(IEnumerable<Task<T>> tasks)
{
    var inputs = tasks.ToList();
    var ce = new CountdownEvent(inputs.Count);
    var tcs = new TaskCompletionSource<T[]>();

    Action<Task> onCompleted = (Task completed) =>
    {
        if (completed.IsFaulted)
            tcs.TrySetException(completed.Exception.InnerExceptions);
        if (ce.Signal() && !tcs.Task.IsCompleted)
            tcs.TrySetResult(inputs.Select(t => t.Result).ToArray());
    }
}

```

```
};

foreach (var t in inputs) t.ContinueWith(onCompleted);
return tcs.Task;
}
```

## Building Task-based Data Structures

In addition to the ability to build custom task-based combinators, having a data structure in [Task](#) and [Task<TResult>](#) that represents both the results of an asynchronous operation and the necessary synchronization to join with it makes it a very powerful type on which to build custom data structures to be used in asynchronous scenarios.

### AsyncCache

One important aspect of a task is that it may be handed out to multiple consumers, all of whom may await it, register continuations with it, get its result or exceptions (in the case of [Task<TResult>](#)), and so on. This makes [Task](#) and [Task<TResult>](#) perfectly suited to be used in an asynchronous caching infrastructure. Here's an example of a small but powerful asynchronous cache built on top of [Task<TResult>](#):

C#

 Copy

```
public class AsyncCache<TKey, TValue>
{
    private readonly Func<TKey, Task<TValue>> _valueFactory;
    private readonly ConcurrentDictionary<TKey, Lazy<Task<TValue>>> _map;

    public AsyncCache(Func<TKey, Task<TValue>> valueFactory)
    {
        if (valueFactory == null) throw new ArgumentNullException("loader");
        _valueFactory = valueFactory;
        _map = new ConcurrentDictionary<TKey, Lazy<Task<TValue>>>();
    }

    public Task<TValue> this[TKey key]
    {
        get
        {
            if (key == null) throw new ArgumentNullException("key");
            return _map.GetOrAdd(key, toAdd =>
                new Lazy<Task<TValue>>(() => _valueFactory(toAdd))).Value;
        }
    }
}
```

The [AsyncCache<TKey,TValue>](#) class accepts as a delegate to its constructor a function that takes a `TKey` and returns a [Task<TResult>](#). Any previously accessed values from the cache are stored in the internal dictionary, and the `AsyncCache` ensures that only one task is generated per key, even if the cache is accessed concurrently.

For example, you can build a cache for downloaded web pages:

C#	 Copy
<pre>private AsyncCache&lt;string,string&gt; m_webPages =     new AsyncCache&lt;string,string&gt;(DownloadStringAsync);</pre>	

You can then use this cache in asynchronous methods whenever you need the contents of a web page. The `AsyncCache` class ensures that you're downloading as few pages as possible, and caches the results.

C#	 Copy
<pre>private async void btnDownload_Click(object sender, RoutedEventArgs e) {     btnDownload.IsEnabled = false;     try     {         txtContents.Text = await m_webPages["https://www.microsoft.com"];     }     finally { btnDownload.IsEnabled = true; } }</pre>	

## AsyncProducerConsumerCollection

You can also use tasks to build data structures for coordinating asynchronous activities. Consider one of the classic parallel design patterns: producer/consumer. In this pattern, producers generate data that is consumed by consumers, and the producers and consumers may run in parallel. For example, the consumer processes item 1, which was previously generated by a producer who is now producing item 2. For the producer/consumer pattern, you invariably need some data structure to store the work created by producers so that the consumers may be notified of new data and find it when available.

Here's a simple data structure built on top of tasks that enables asynchronous methods to be used as producers and consumers:

C#



```
public class AsyncProducerConsumerCollection<T>
{
    private readonly Queue<T> m_collection = new Queue<T>();
    private readonly Queue<TaskCompletionSource<T>> m_waiting =
        new Queue<TaskCompletionSource<T>>();

    public void Add(T item)
    {
        TaskCompletionSource<T> tcs = null;
        lock (m_collection)
        {
            if (m_waiting.Count > 0) tcs = m_waiting.Dequeue();
            else m_collection.Enqueue(item);
        }
        if (tcs != null) tcs.TrySetResult(item);
    }

    public Task<T> Take()
    {
        lock (m_collection)
        {
            if (m_collection.Count > 0)
            {
                return Task.FromResult(m_collection.Dequeue());
            }
            else
            {
                var tcs = new TaskCompletionSource<T>();
                m_waiting.Enqueue(tcs);
                return tcs.Task;
            }
        }
    }
}
```

With that data structure in place, you can write code such as the following:

C#



```
private static AsyncProducerConsumerCollection<int> m_data = ...;
...
private static async Task ConsumerAsync()
{
    while(true)
    {
        int nextItem = await m_data.Take();
        ProcessNextItem(nextItem);
    }
}
```

```
    }  
}  
...  
private static void Produce(int data)  
{  
    m_data.Add(data);  
}
```

The [System.Threading.Tasks.Dataflow](#) namespace includes the [BufferBlock<T>](#) type, which you can use in a similar manner, but without having to build a custom collection type:

C#

 Copy

```
private static BufferBlock<int> m_data = ...;  
...  
private static async Task ConsumerAsync()  
{  
    while(true)  
    {  
        int nextItem = await m_data.ReceiveAsync();  
        ProcessNextItem(nextItem);  
    }  
}  
...  
private static void Produce(int data)  
{  
    m_data.Post(data);  
}
```

### ⓘ Note

The [System.Threading.Tasks.Dataflow](#) namespace is available in the .NET Framework 4.5 through **NuGet**. To install the assembly that contains the [System.Threading.Tasks.Dataflow](#) namespace, open your project in Visual Studio, choose **Manage NuGet Packages** from the Project menu, and search online for the Microsoft.Tpl.Dataflow package.

## See also

- [Task-based Asynchronous Pattern \(TAP\)](#)
- [Implementing the Task-based Asynchronous Pattern](#)
- [Interop with Other Asynchronous Patterns and Types](#)



Is this page helpful?

 Yes  No

---