Table of contents

- Timer callbacks
- Implicit async void delegates
- ConcurrentDictionary.GetOrAdd
- Constructors
- WindowsIdentity.RunImpersonated

Asynchronous Programming

Asynchronous programming has been around for several years on the .NET platform but has historically been very difficult to do well. Since the introduction of async/await in C# 5 asynchronous programming has become mainstream. Modern frameworks (like ASP.NET Core) are fully asynchronous and it's very hard to avoid the async keyword when writing web services. As a result, there's been lots of confusion on the best practices for async and how to use it properly. This section will try to lay out some guidance with examples of bad and good patterns of how to write asynchronous code.

Asynchrony is viral

Once you go async, all of your callers **SHOULD** be async, since efforts to be async amount to nothing unless the entire callstack is async. In many cases, being partially async can be worse than being entirely synchronous. Therefore it is best to go all in, and make everything async at once.

❌ **BAD** This example uses the `Task.Result` and as a result blocks the current thread to wait for the result. This is an example of sync over async.

```
public int DoSomethingAsync()
{
    var result = CallDependencyAsync().Result;
    return result + 1;
}
```

✅ **GOOD** This example uses the await keyword to get the result from `CallDependencyAsync`.

```
public async Task<int> DoSomethingAsync()
{
    var result = await CallDependencyAsync();
    return result + 1;
}
```

Async void

Use of async void in ASP.NET Core applications is **ALWAYS** bad. Avoid it, never do it. Typically, it's used when developers are trying to implement fire and forget patterns triggered by a controller action. Async void methods will crash the process if an exception is thrown. We'll look at more of the patterns that cause developers to do this in ASP.NET Core applications but here's a simple example:

❌ **BAD** Async void methods can't be tracked and therefore unhandled exceptions can result in application crashes.

```
public class MyController : Controller
{
    [HttpPost("/start")]
    public IActionResult Post()
    {
        BackgroundOperationAsync();
        return Accepted();
    }

    public async void BackgroundOperationAsync()
    {
        var result = await CallDependencyAsync();
        DoSomething(result);
    }
}
```

✅ **GOOD** `Task`-returning methods are better since unhandled exceptions trigger the `TaskScheduler.UnobservedTaskException`.

```
public class MyController : Controller
{
    [HttpPost("/start")]
    public IActionResult Post()
    {
        Task.Run(BackgroundOperationAsync);
        return Accepted();
    }

    public async Task BackgroundOperationAsync()
    {
        var result = await CallDependencyAsync();
        DoSomething(result);
    }
}
```

Prefer `Task.FromResult` over `Task.Run` for pre-computed or trivially computed data

For pre-computed results, there's no need to call `Task.Run`, that will end up queuing a work item to the thread pool that will immediately complete with the pre-computed value. Instead, use `Task.FromResult`, to create a task wrapping already computed data.

❌ **BAD** This example wastes a thread-pool thread to return a trivially computed value.

```
public class MyLibrary
{
   public Task<int> AddAsync(int a, int b)
   {
       return Task.Run(() => a + b);
   }
}
```

✅ **GOOD** This example uses `Task.FromResult` to return the trivially computed value. It does not use any extra threads as a result.

```
public class MyLibrary
{
   public Task<int> AddAsync(int a, int b)
   {
       return Task.FromResult(a + b);
   }
}
```

💡 **NOTE: Using `Task.FromResult` will result in a `Task` allocation. Using `ValueTask<T>` can completely remove that allocation.**

✅ **GOOD** This example uses a `ValueTask<int>` to return the trivially computed value. It does not use any extra threads as a result. It also does not allocate an object on the managed heap.

```
public class MyLibrary
{
   public ValueTask<int> AddAsync(int a, int b)
   {
       return new ValueTask<int>(a + b);
```

```
    }
}
```

Avoid using Task.Run for long running work that blocks the thread

Long running work in this context refers to a thread that's running for the lifetime of the application doing background work (like processing queue items, or sleeping and waking up to process some data). `Task.Run` will queue a work item to the thread pool. The assumption is that that work will finish quickly (or quickly enough to allow reusing that thread within some reasonable timeframe). Stealing a thread-pool thread for long-running work is bad since it takes that thread away from other work that could be done (timer callbacks, task continuations etc). Instead, spawn a new thread manually to do long running blocking work.

💡 **NOTE: The thread pool grows if you block threads but it's bad practice to do so.**

💡 **NOTE: `Task.Factory.StartNew` has an option `TaskCreationOptions.LongRunning` that under the covers creates a new thread and returns a Task that represents the execution. Using this properly requires several non-obvious parameters to be passed in to get the right behavior on all platforms.**

💡 **NOTE: Don't use `TaskCreationOptions.LongRunning` with async code as this will create a new thread which will be destroyed after first `await`.**

❌ **BAD** This example steals a thread-pool thread forever, to execute queued work on a `BlockingCollection<T>`.

```
public class QueueProcessor
{
    private readonly BlockingCollection<Message> _messageQueue = new BlockingCollect

    public void StartProcessing()
    {
        Task.Run(ProcessQueue);
    }

    public void Enqueue(Message message)
    {
        _messageQueue.Add(message);
    }

    private void ProcessQueue()
    {
```

```
        foreach (var item in _messageQueue.GetConsumingEnumerable())
        {
            ProcessItem(item);
        }
    }

    private void ProcessItem(Message message) { }
}
```

✔️ **GOOD** This example uses a dedicated thread to process the message queue instead of a thread-pool thread.

```
public class QueueProcessor
{
    private readonly BlockingCollection<Message> _messageQueue = new BlockingCollect

    public void StartProcessing()
    {
        var thread = new Thread(ProcessQueue)
        {
            // This is important as it allows the process to exit while this thread
            IsBackground = true
        };
        thread.Start();
    }

    public void Enqueue(Message message)
    {
        _messageQueue.Add(message);
    }

    private void ProcessQueue()
    {
        foreach (var item in _messageQueue.GetConsumingEnumerable())
        {
            ProcessItem(item);
        }
    }

    private void ProcessItem(Message message) { }
}
```

Avoid using `Task.Result` and `Task.Wait`

There are very few ways to use `Task.Result` and `Task.Wait` correctly so the general advice is to completely avoid using them in your code.

⚠ Sync over `async`

Using `Task.Result` or `Task.Wait` to block wait on an asynchronous operation to complete is *MUCH* worse than calling a truly synchronous API to block. This phenomenon is dubbed "Sync over async". Here is what happens at a very high level:

- An asynchronous operation is kicked off.
- The calling thread is blocked waiting for that operation to complete.
- When the asynchronous operation completes, it unblocks the code waiting on that operation. This takes place on another thread.

The result is that we need to use 2 threads instead of 1 to complete synchronous operations. This usually leads to thread-pool starvation and results in service outages.

⚠ Deadlocks

The `SynchronizationContext` is an abstraction that gives application models a chance to control where asynchronous continuations run. ASP.NET (non-core), WPF and Windows Forms each have an implementation that will result in a deadlock if Task.Wait or Task.Result is used on the main thread. This behavior has led to a bunch of "clever" code snippets that show the "right" way to block waiting for a Task. The truth is, there's no good way to block waiting for a Task to complete.

💡**NOTE: ASP.NET Core does not have a `SynchronizationContext` and is not prone to the deadlock problem.**

❌ **BAD** The below are all examples that are, in one way or another, trying to avoid the deadlock situation but still succumb to "sync over async" problems.

```
public string DoOperationBlocking()
{
    // Bad - Blocking the thread that enters.
    // DoAsyncOperation will be scheduled on the default task scheduler, and remove
    // In the case of an exception, this method will throw an AggregateException wra
    return Task.Run(() => DoAsyncOperation()).Result;
}

public string DoOperationBlocking2()
{
    // Bad - Blocking the thread that enters.
    // DoAsyncOperation will be scheduled on the default task scheduler, and remove
```

```
    return Task.Run(() => DoAsyncOperation()).GetAwaiter().GetResult();
}

public string DoOperationBlocking3()
{
    // Bad - Blocking the thread that enters, and blocking the theadpool thread insi
    // In the case of an exception, this method will throw an AggregateException con
    return Task.Run(() => DoAsyncOperation().Result).Result;
}

public string DoOperationBlocking4()
{
    // Bad - Blocking the thread that enters, and blocking the theadpool thread insi
    return Task.Run(() => DoAsyncOperation().GetAwaiter().GetResult()).GetAwaiter().
}

public string DoOperationBlocking5()
{
    // Bad - Blocking the thread that enters.
    // Bad - No effort has been made to prevent a present SynchonizationContext from
    // In the case of an exception, this method will throw an AggregateException wra
    return DoAsyncOperation().Result;
}

public string DoOperationBlocking6()
{
    // Bad - Blocking the thread that enters.
    // Bad - No effort has been made to prevent a present SynchonizationContext from
    return DoAsyncOperation().GetAwaiter().GetResult();
}

public string DoOperationBlocking7()
{
    // Bad - Blocking the thread that enters.
    // Bad - No effort has been made to prevent a present SynchonizationContext from
    var task = DoAsyncOperation();
    task.Wait();
    return task.GetAwaiter().GetResult();
}
```

Prefer `await` over `ContinueWith`

`Task` existed before the async/await keywords were introduced and as such provided ways to execute continuations without relying on the language. Although these methods are still valid to use, we generally recommend that you prefer `async` / `await` to using `ContinueWith`. `ContinueWith` also does not capture the `SynchronizationContext` and as a result is actually semantically different to `async` / `await`.

❌ **BAD** The example uses `ContinueWith` instead of `async`

```
public Task<int> DoSomethingAsync()
{
    return CallDependencyAsync().ContinueWith(task =>
    {
        return task.Result + 1;
    });
}
```

✔️ **GOOD** This example uses the `await` keyword to get the result from `CallDependencyAsync`.

```
public async Task<int> DoSomethingAsync()
{
    var result = await CallDependencyAsync();
    return result + 1;
}
```

Always create `TaskCompletionSource<T>` with `TaskCreationOptions.RunContinuationsAsynchronously`

`TaskCompletionSource<T>` is an important building block for libraries trying to adapt things that are not inherently awaitable to be awaitable via a `Task`. It is also commonly used to build higher-level operations (such as batching and other combinators) on top of existing asynchronous APIs. By default, `Task` continuations will run *inline* on the same thread that calls Try/Set(Result/Exception/Canceled). As a library author, this means having to understand that calling code can resume directly on your thread. This is extremely dangerous and can result in deadlocks, thread-pool starvation, corruption of state (if code runs unexpectedly) and more.

Always use `TaskCreationOptions.RunContinuationsAsynchronously` when creating the `TaskCompletionSource<T>`. This will dispatch the continuation onto the thread pool instead of executing it inline.

❌ **BAD** This example does not use `TaskCreationOptions.RunContinuationsAsynchronously` when creating the `TaskCompletionSource<T>`.

```
public Task<int> DoSomethingAsync()
{
    var tcs = new TaskCompletionSource<int>();

    var operation = new LegacyAsyncOperation();
    operation.Completed += result =>
    {
        // Code awaiting on this task will resume on this thread!
        tcs.SetResult(result);
    };

    return tcs.Task;
}
```

✔ **GOOD** This example uses `TaskCreationOptions.RunContinuationsAsynchronously` when creating the `TaskCompletionSource<T>`.

```
public Task<int> DoSomethingAsync()
{
    var tcs = new TaskCompletionSource<int>(TaskCreationOptions.RunContinuationsAsyn

    var operation = new LegacyAsyncOperation();
    operation.Completed += result =>
    {
        // Code awaiting on this task will resume on a different thread-pool thread
        tcs.SetResult(result);
    };

    return tcs.Task;
}
```

💡 **NOTE: There are 2 enums that look alike.** **`TaskCreationOptions.RunContinuationsAsynchronously`** **and** **`TaskContinuationOptions.RunContinuationsAsynchronously`** **. Be careful not to confuse their usage.**

Always dispose `CancellationTokenSource` (s) used for timeouts

`CancellationTokenSource` objects that are used for timeouts (are created with timers or uses the `CancelAfter` method), can put pressure on the timer queue if not disposed.

❌ **BAD** This example does not dispose the `CancellationTokenSource` and as a result the timer stays in the queue for 10 seconds after each request is made.

```
public async Task<Stream> HttpClientAsyncWithCancellationBad()
{
    var cts = new CancellationTokenSource(TimeSpan.FromSeconds(10));

    using (var client = _httpClientFactory.CreateClient())
    {
        var response = await client.GetAsync("http://backend/api/1", cts.Token);
        return await response.Content.ReadAsStreamAsync();
    }
}
```

✅ **GOOD** This example disposes the `CancellationTokenSource` and properly removes the timer from the queue.

```
public async Task<Stream> HttpClientAsyncWithCancellationGood()
{
    using (var cts = new CancellationTokenSource(TimeSpan.FromSeconds(10)))
    {
        using (var client = _httpClientFactory.CreateClient())
        {
            var response = await client.GetAsync("http://backend/api/1", cts.Token);
            return await response.Content.ReadAsStreamAsync();
        }
    }
}
```

Always flow `CancellationToken`(s) to APIs that take a `CancellationToken`

Cancellation is cooperative in .NET. Everything in the call-chain has to be explicitly passed the `CancellationToken` in order for it to work well. This means you need to explicitly pass the token into other APIs that take a token if you want cancellation to be most effective.

❌ **BAD** This example neglects to pass the `CancellationToken` to `Stream.ReadAsync` making the operation effectively not cancellable.

```
public async Task<string> DoAsyncThing(CancellationToken cancellationToken = default
{
   byte[] buffer = new byte[1024];
```

```
    // We forgot to pass flow cancellationToken to ReadAsync
    int read = await _stream.ReadAsync(buffer, 0, buffer.Length);
    return Encoding.UTF8.GetString(buffer, 0, read);
}
```

✔ **GOOD** This example passes the `CancellationToken` into `Stream.ReadAsync`.

```
public async Task<string> DoAsyncThing(CancellationToken cancellationToken = default
{
    byte[] buffer = new byte[1024];
    // This properly flows cancellationToken to ReadAsync
    int read = await _stream.ReadAsync(buffer, 0, buffer.Length, cancellationToken);
    return Encoding.UTF8.GetString(buffer, 0, read);
}
```

Cancelling uncancellable operations

One of the coding patterns that appears when doing asynchronous programming is cancelling an uncancellable operation. This usually means creating another task that completes when a timeout or `CancellationToken` fires, and then using `Task.WhenAny` to detect a complete or cancelled operation.

Using CancellationTokens

✘ **BAD** This example uses `Task.Delay(-1, token)` to create a `Task` that completes when the `CancellationToken` fires, but if it doesn't fire, there's no way to dispose the `CancellationTokenRegistration`. This can lead to a memory leak.

```
public static async Task<T> WithCancellation<T>(this Task<T> task, CancellationToken
{
    // There's no way to dispose the registration
    var delayTask = Task.Delay(-1, cancellationToken);

    var resultTask = await Task.WhenAny(task, delayTask);
    if (resultTask == delayTask)
    {
        // Operation cancelled
        throw new OperationCanceledException();
    }

    return await task;
}
```

✔️ **GOOD** This example disposes the `CancellationTokenRegistration` when one of the `Task(s)` complete.

```
public static async Task<T> WithCancellation<T>(this Task<T> task, CancellationToken
{
    var tcs = new TaskCompletionSource<object>(TaskCreationOptions.RunContinuationsA

    // This disposes the registration as soon as one of the tasks trigger
    using (cancellationToken.Register(state =>
    {
        ((TaskCompletionSource<object>)state).TrySetResult(null);
    },
    tcs))
    {
        var resultTask = await Task.WhenAny(task, tcs.Task);
        if (resultTask == tcs.Task)
        {
            // Operation cancelled
            throw new OperationCanceledException(cancellationToken);
        }

        return await task;
    }
}
```

Using a timeout

❌ **BAD** This example does not cancel the timer even if the operation successfuly completes. This means you could end up with lots of timers, which can flood the timer queue.

```
public static async Task<T> TimeoutAfter<T>(this Task<T> task, TimeSpan timeout)
{
    var delayTask = Task.Delay(timeout);

    var resultTask = await Task.WhenAny(task, delayTask);
    if (resultTask == delayTask)
    {
        // Operation cancelled
        throw new OperationCanceledException();
    }

    return await task;
}
```

✔️ **GOOD** This example cancels the timer if the operation succesfully completes.

```
public static async Task<T> TimeoutAfter<T>(this Task<T> task, TimeSpan timeout)
{
    using (var cts = new CancellationTokenSource())
    {
        var delayTask = Task.Delay(timeout, cts.Token);

        var resultTask = await Task.WhenAny(task, delayTask);
        if (resultTask == delayTask)
        {
            // Operation cancelled
            throw new OperationCanceledException();
        }
        else
        {
            // Cancel the timer task so that it does not fire
            cts.Cancel();
        }

        return await task;
    }
}
```

Always call `FlushAsync` on `StreamWriter`(s) or `Stream`(s) before calling `Dispose`

When writing to a `Stream` or `StreamWriter`, even if the asynchronous overloads are used for writing, the underlying data might be buffered. When data is buffered, disposing the `Stream` or `StreamWriter` via the `Dispose` method will synchronously write/flush, which results in blocking the thread and could lead to thread-pool starvation. Either use the asynchronous `DisposeAsync` method (for example via `await using`) or call `FlushAsync` before calling `Dispose`.

💡 **NOTE: This is only problematic if the underlying subsystem does IO.**

❌ **BAD** This example ends up blocking the request by writing synchronously to the HTTP-response body.

```
app.Run(async context =>
{
    // The implicit Dispose call will synchronously write to the response body
    using (var streamWriter = new StreamWriter(context.Response.Body))
    {
```

```
        await streamWriter.WriteAsync("Hello World");
    }
});
```

✔️ **GOOD** This example asynchronously flushes any buffered data while disposing the `StreamWriter`.

```
app.Run(async context =>
{
    // The implicit AsyncDispose call will flush asynchronously
    await using (var streamWriter = new StreamWriter(context.Response.Body))
    {
        await streamWriter.WriteAsync("Hello World");
    }
});
```

✔️ **GOOD** This example asynchronously flushes any buffered data before disposing the `StreamWriter`.

```
app.Run(async context =>
{
    using (var streamWriter = new StreamWriter(context.Response.Body))
    {
        await streamWriter.WriteAsync("Hello World");

        // Force an asynchronous flush
        await streamWriter.FlushAsync();
    }
});
```

Prefer `async` / `await` over directly returning `Task`

There are benefits to using the `async` / `await` keyword instead of directly returning the `Task`:

- Asynchronous and synchronous exceptions are normalized to always be asynchronous.
- The code is easier to modify (consider adding a `using`, for example).
- Diagnostics of asynchronous methods are easier (debugging hangs etc).
- Exceptions thrown will be automatically wrapped in the returned `Task` instead of surprising the caller with an actual exception.

❌ **BAD** This example directly returns the `Task` to the caller.

```
public Task<int> DoSomethingAsync()
{
    return CallDependencyAsync();
}
```

✔️ **GOOD** This examples uses async/await instead of directly returning the Task.

```
public async Task<int> DoSomethingAsync()
{
    return await CallDependencyAsync();
}
```

💡**NOTE: There are performance considerations when using an async state machine over directly returning the `Task`. It's always faster to directly return the `Task` since it does less work but you end up changing the behavior and potentially losing some of the benefits of the async state machine.**

ConfigureAwait

TBD

Scenarios

The above tries to distill general guidance, but doesn't do justice to the kinds of real-world situations that cause code like this to be written in the first place (bad code). This section tries to take concrete examples from real applications and turn them into something simple to help you relate these problems to existing codebases.

`Timer` callbacks

❌ **BAD** The `Timer` callback is `void`-returning and we have asynchronous work to execute. This example uses `async void` to accomplish it and as a result can crash the process if an exception occurs.

```
public class Pinger
{
    private readonly Timer _timer;
    private readonly HttpClient _client;

    public Pinger(HttpClient client)
```

```
    {
        _client = client;
        _timer = new Timer(Heartbeat, null, 1000, 1000);
    }

    public async void Heartbeat(object state)
    {
        await _client.GetAsync("http://mybackend/api/ping");
    }
}
```

❌ **BAD** This attempts to block in the `Timer` callback. This may result in thread-pool starvation and is an example of sync over async

```
public class Pinger
{
    private readonly Timer _timer;
    private readonly HttpClient _client;

    public Pinger(HttpClient client)
    {
        _client = client;
        _timer = new Timer(Heartbeat, null, 1000, 1000);
    }

    public void Heartbeat(object state)
    {
        _client.GetAsync("http://mybackend/api/ping").GetAwaiter().GetResult();
    }
}
```

✔️ **GOOD** This example uses an `async Task`-based method and discards the `Task` in the `Timer` callback. If this method fails, it will not crash the process. Instead, it will fire the `TaskScheduler.UnobservedTaskException` event.

```
public class Pinger
{
    private readonly Timer _timer;
    private readonly HttpClient _client;

    public Pinger(HttpClient client)
    {
        _client = client;
        _timer = new Timer(Heartbeat, null, 1000, 1000);
    }

    public void Heartbeat(object state)
```

```
    {
        // Discard the result
        _ = DoAsyncPing();
    }

    private async Task DoAsyncPing()
    {
        await _client.GetAsync("http://mybackend/api/ping");
    }
}
```

Implicit `async void` delegates

Imagine a `BackgroundQueue` with a `FireAndForget` that takes a callback. This method will execute the callback at some time in the future.

❌ **BAD** This will force callers to either block in the callback or use an `async void` delegate.

```
public class BackgroundQueue
{
    public static void FireAndForget(Action action) { }
}
```

❌ **BAD** This calling code is creating an `async void` method implicitly. The compiler fully supports this today.

```
public class Program
{
    public void Main(string[] args)
    {
        var httpClient = new HttpClient();
        BackgroundQueue.FireAndForget(async () =>
        {
            await httpClient.GetAsync("http://pinger/api/1");
        });

        Console.ReadLine();
    }
}
```

✅ **GOOD** This BackgroundQueue implementation offers both sync and `async` callback overloads.

```
public class BackgroundQueue
{
    public static void FireAndForget(Action action) { }
    public static void FireAndForget(Func<Task> action) { }
}
```

ConcurrentDictionary.GetOrAdd

It's pretty common to cache the result of an asynchronous operation and `ConcurrentDictionary` is a good data structure for doing that. `GetOrAdd` is a convenience API for trying to get an item if it's already there or adding it if it isn't. The callback is synchronous so it's tempting to write code that uses `Task.Result` to produce the value of an asynchronous process but that can lead to thread-pool starvation.

❌ **BAD** This may result in thread-pool starvation since we're blocking the request thread if the person data is not cached.

```
public class PersonController : Controller
{
   private AppDbContext _db;

   // This cache needs expiration
   private static ConcurrentDictionary<int, Person> _cache = new ConcurrentDictionar

   public PersonController(AppDbContext db)
   {
      _db = db;
   }

   public IActionResult Get(int id)
   {
       var person = _cache.GetOrAdd(id, (key) => _db.People.FindAsync(key).Result);
       return Ok(person);
   }
}
```

✔️ **GOOD** This implementation won't result in thread-pool starvation since we're storing a task instead of the result itself.

⚠ `ConcurrentDictionary.GetOrAdd` will potentially run the cache callback multiple times in parallel. This can result in kicking off expensive computations multiple times.

```
public class PersonController : Controller
{
   private AppDbContext _db;

   // This cache needs expiration
   private static ConcurrentDictionary<int, Task<Person>> _cache = new ConcurrentDic

   public PersonController(AppDbContext db)
   {
      _db = db;
   }

   public async Task<IActionResult> Get(int id)
   {
       var person = await _cache.GetOrAdd(id, (key) => _db.People.FindAsync(key));
       return Ok(person);
   }
}
```

✅ **GOOD** This implementation fixes the multiple-executing callback issue by using the `async` lazy pattern.

```
public class PersonController : Controller
{
   private AppDbContext _db;

   // This cache needs expiration
   private static ConcurrentDictionary<int, AsyncLazy<Person>> _cache = new Concurre

   public PersonController(AppDbContext db)
   {
      _db = db;
   }

   public async Task<IActionResult> Get(int id)
   {
       var person = await _cache.GetOrAdd(id, (key) => new AsyncLazy<Person>(() => _
       return Ok(person);
   }

   private class AsyncLazy<T> : Lazy<Task<T>>
   {
      public AsyncLazy(Func<Task<T>> valueFactory) : base(valueFactory)
      {
      }
   }
}
```

Constructors

Constructors are synchronous. If you need to initialize some logic that may be asynchronous, there are a couple of patterns for dealing with this.

Here's an example of using a client API that needs to connect asynchronously before use.

```
public interface IRemoteConnectionFactory
{
   Task<IRemoteConnection> ConnectAsync();
}

public interface IRemoteConnection
{
    Task PublishAsync(string channel, string message);
    Task DisposeAsync();
}
```

❌ **BAD** This example uses `Task.Result` to get the connection in the constructor. This could lead to thread-pool starvation and deadlocks.

```
public class Service : IService
{
    private readonly IRemoteConnection _connection;

    public Service(IRemoteConnectionFactory connectionFactory)
    {
        _connection = connectionFactory.ConnectAsync().Result;
    }
}
```

✅ **GOOD** This implementation uses a static factory pattern in order to allow asynchronous construction:

```
public class Service : IService
{
    private readonly IRemoteConnection _connection;

    private Service(IRemoteConnection connection)
    {
        _connection = connection;
    }
```

```
    public static async Task<Service> CreateAsync(IRemoteConnectionFactory connectio
    {
        return new Service(await connectionFactory.ConnectAsync());
    }
}
```

WindowsIdentity.RunImpersonated

This API runs the specified action as the impersonated Windows identity. An asynchronous version of the callback was introduced in .NET 5.0.

❌ **BAD** This example tries to execute the query asynchronously, and then wait for it outside of the call to `RunImpersonated`. This will throw because the query might be executing outside of the impersonation context.

```
public async Task<IEnumerable<Product>> GetDataImpersonatedAsync(SafeAccessTokenHand
{
    Task<IEnumerable<Product>> products = null;
    WindowsIdentity.RunImpersonated(
        safeAccessTokenHandle,
        context =>
        {
            products = _db.QueryAsync("SELECT Name from Products");
        }};
    return await products;
}
```

❌ **BAD** This example uses `Task.Result` to get the connection in the constructor. This could lead to thread-pool starvation and deadlocks.

```
public IEnumerable<Product> GetDataImpersonatedAsync(SafeAccessTokenHandle safeAcces
{
    return WindowsIdentity.RunImpersonated(
        safeAccessTokenHandle,
        context => _db.QueryAsync("SELECT Name from Products").Result);
}
```

✅ **GOOD** This example awaits the result of `RunImpersonated` (the delegate is `Func<Task<IEnumerable<Product>>>` in this case). It is the recommended practice in framewroks earlier than .NET 5.0.

```
public async Task<IEnumerable<Product>> GetDataImpersonatedAsync(SafeAccessTokenHand
{
    return await WindowsIdentity.RunImpersonated(
        safeAccessTokenHandle,
        context => _db.QueryAsync("SELECT Name from Products"));
}
```

✅ **GOOD** This example uses the asynchronous `RunImparsonatedAsync` function and awaits its result. It is available in .NET 5.0 or newer.

```
public async Task<IEnumerable<Product>> GetDataImpersonatedAsync(SafeAccessTokenHand
{
    return await WindowsIdentity.RunImpersonatedAsync(
        safeAccessTokenHandle,
        context => _db.QueryAsync("SELECT Name from Products"));
}
```