Asynchronous programming in .NET

by Gerardo Lijs

Twitter @GerardoLijs

C# MeetUp Barcelona

April 2019

Content

What is Asynchronous Programming

CPU-bound vs IO-bound

Obsolete patterns

Tasks and async/await

Practical use cases

- Responsive user interfaces (WPF, WinForms, etc)
- Scalability and performance in web applications

CPU-bound operations

Synchronous method

```
int GetPrimesCount(int start, int count)
{
    return ParallelEnumerable.Range(start, count).Count(n =>
              Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0));
}
```

Asynchronous method

```
Task<int> GetPrimesCountAsync(int start, int count)
{
    return Task.Run(() =>
       ParallelEnumerable.Range(start, count).Count(n =>
         Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0)));
}
```

- Use Task.Run to execute expensive CPU-bound code in the ThreadPool
- Define a Task returning method
- In general you will not need other methods of TaskFactory class

IO-bound operations

- Hard drive
- Network
- Database
- API calls
- Hardware read/write (serial port, camera, sensor, etc)

Unless you work in very specific industries, 95% or more of the code you write will never be CPU-bound. Most of the time you will need asynchronous to deal with IO-bound operations and the times you will use CPU-bound operations you will probably use a library that provides asynchronous methods for you to interact with.

Responsive user interfaces

Demo of a WPF desktop application

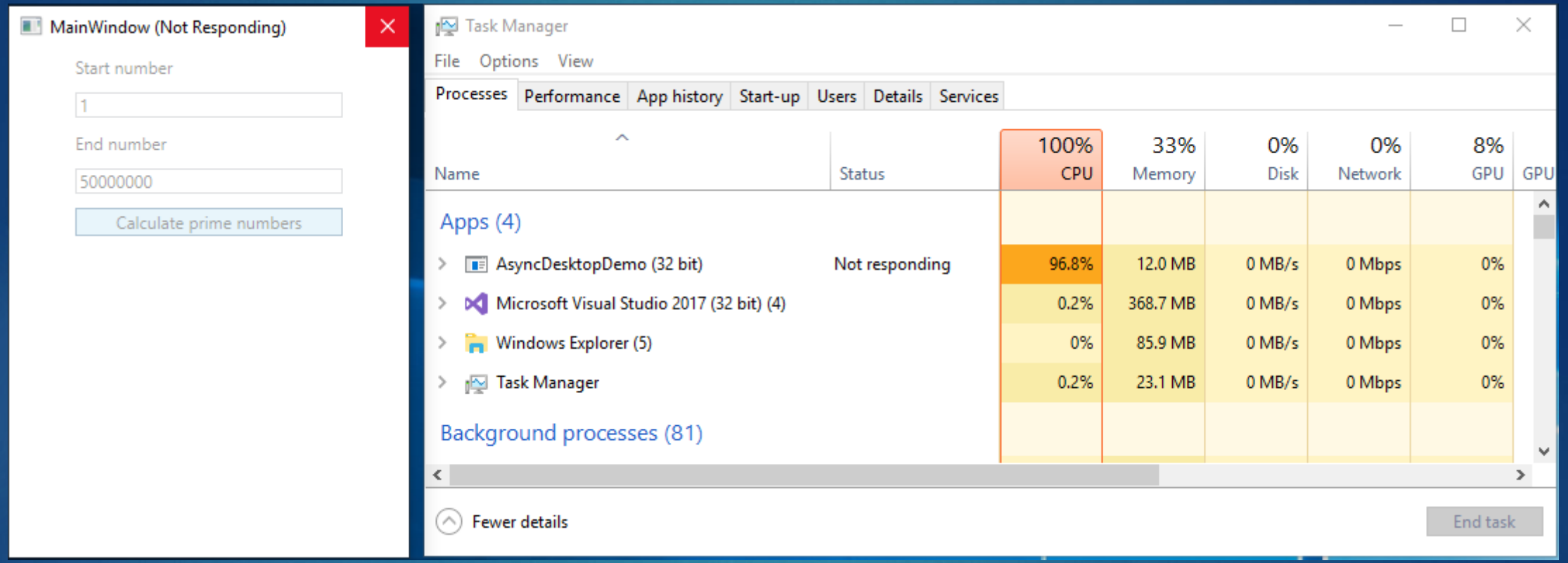
- Targets .NET Core 3 Preview4 but works exactly the same with .NET 4.7.2 or earlier
- The behaviour is the same in WinForms

You will need https://dotnet.microsoft.com/download/dotnet-core/3.0

and VS 2019 enabled for .NET Core

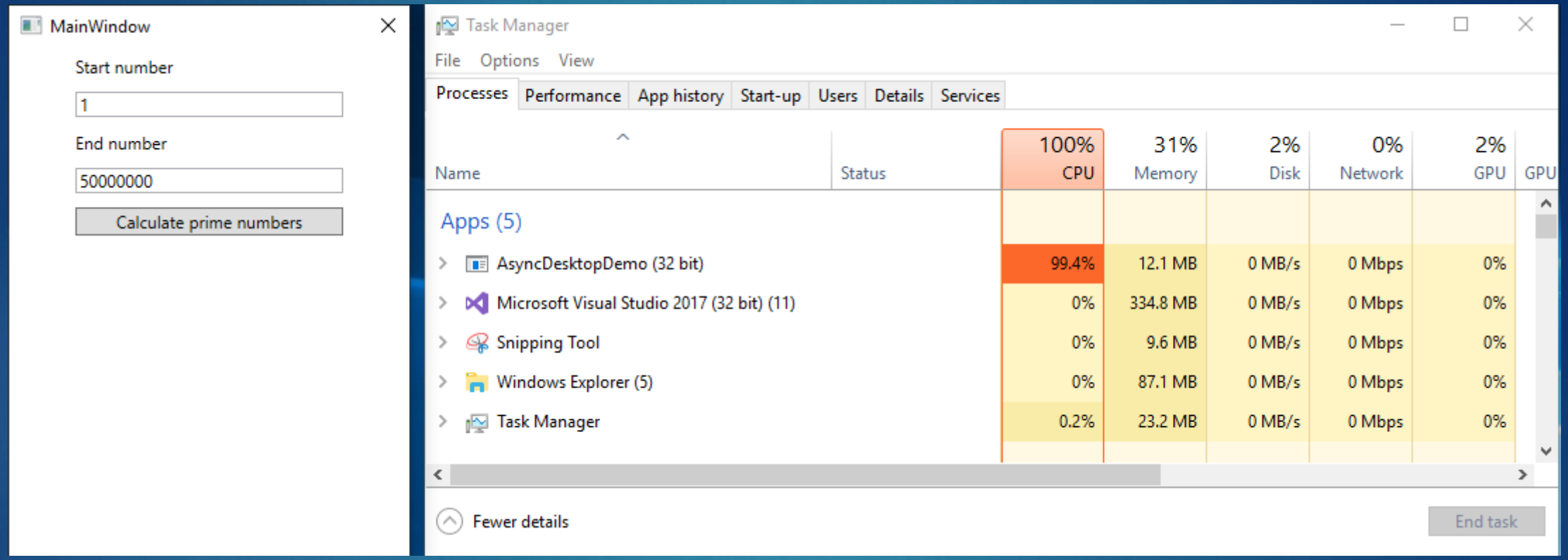
https://visualstudiomagazine.com/articles/2019/03/08/vs-2019-core-tip.aspx

Responsive user interfaces



This is bad! Application is not responding

Responsive user interfaces



This is good! Application still responsive with 100% CPU use

Responsive user interfaces

Sync code - Before

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    int.TryParse(StartNumberTextBox.Text, out int start);
    int.TryParse(EndNumberTextBox.Text, out int end);

    ResultTextBlock.Text = "";
    int result = GetPrimesCount(start, end);
    ResultTextBlock.Text = $"{result} prime numbers between {start} and {end}";
}

private int GetPrimesCount(int start, int count)
{
    return ParallelEnumerable.Range(start, count).Count(n => Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0));
}
```

Async code - After

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    int.TryParse(StartNumberTextBox.Text, out int start);
    int.TryParse(EndNumberTextBox.Text, out int end);

    ResultTextBlock.Text = "";
    int result = await GetPrimesCount(start, end);
    ResultTextBlock.Text = $"{result} prime numbers between {start} and {end}";
}

private Task<int> GetPrimesCount(int start, int count)
{
    return Task.Run(() =>
        ParallelEnumerable.Range(start, count).Count(n => Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0)));
}
```

Responsive user interfaces

Summary

- Use async with IO-bound operations whenever possible
- Don't block!
- Use async for expensive CPU-bound operations
- Use progress report and progress dialogs for better user experience
- Allow user to cancel operations when possible
- Use benchmarking tools such as BenchmarkDotNet (https://github.com/dotnet/BenchmarkDotNet)

Source code

https://github.com/gerardo-lijs/CSharpMeetup-Mar2019

Responsive user interfaces

Demo of a Web API running IO-bound operation

- Targets .NET Core 3 Preview4 but works exactly the same with .NET Core 2
- The behaviour is the same in Web API using .NET 4.7.2 or earlier

Source code

https://github.com/gerardo-lijs/CSharpMeetup-Mar2019

Scalability and performance in web applications

Simple API method that downloads a web page using synchronous code

```
[Route("api/[controller]")]
[ApiController]
public class IOBoundController : ControllerBase
{
    private string GetHtml()
    {
        var client = new System.Net.WebClient();
        string response = client.DownloadString("https://www.dotnetfoundation.org");
        return response;
    }

    // GET api/iobound
    // .\bombardier.exe "http://localhost:5000/api/iobound" -n 20 -t 100s
    [HttpGet]
    public ActionResult<int> Get()
    {
        var webContent = GetHtml();
        return webContent.Length;
    }
```

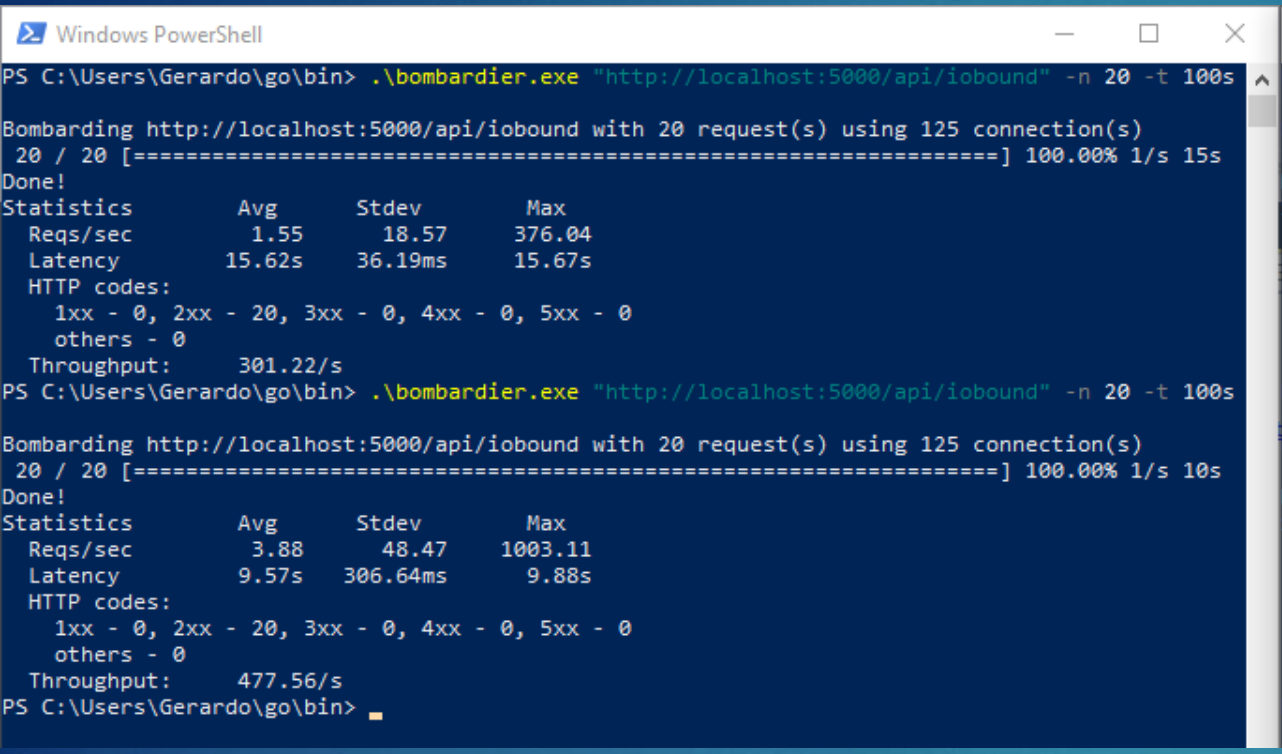
Scalability and performance in web applications

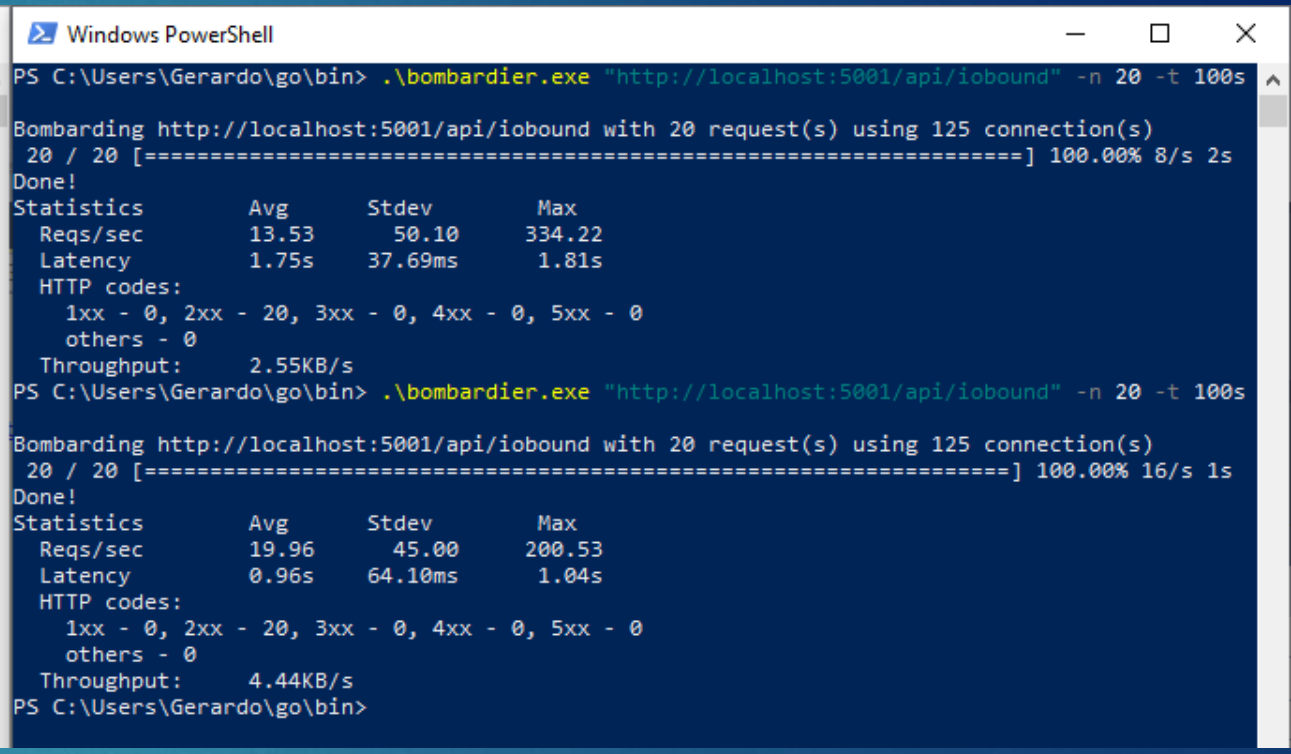
Same method refactored to use asynchronous code

```
[Route("api/[controller]")]
[ApiController]
public class IOBoundController : ControllerBase
{
    private async Task<string> GetHtml()
    {
        var client = new System.Net.Http.HttpClient();
        string response = await client.GetStringAsync("https://www.dotnetfoundation.org");
        return response;
    }

    // GET api/iobound
    // .\bombardier.exe "http://localhost:5001/api/iobound" -n 20 -t 100s
    [HttpGet]
    public async Task<ActionResult<int>> Get()
    {
        var webContent = await GetHtml();
        return webContent.Length;
    }
```

Scalability and performance in web applications





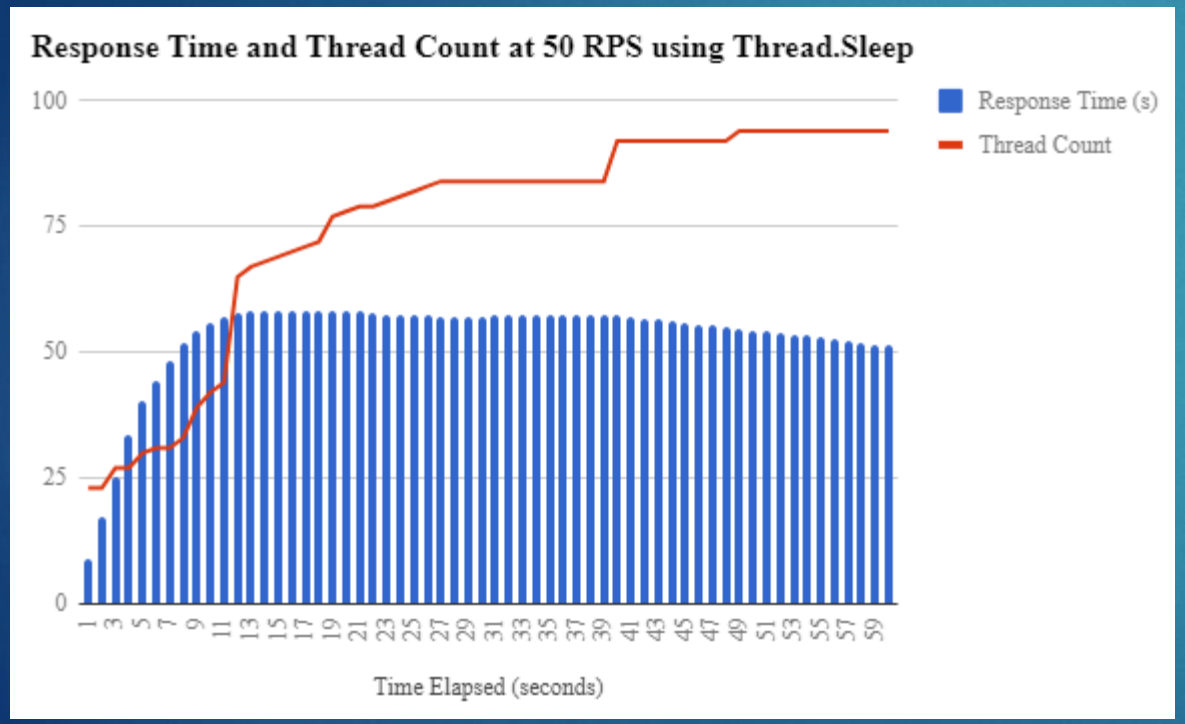
This is bad!

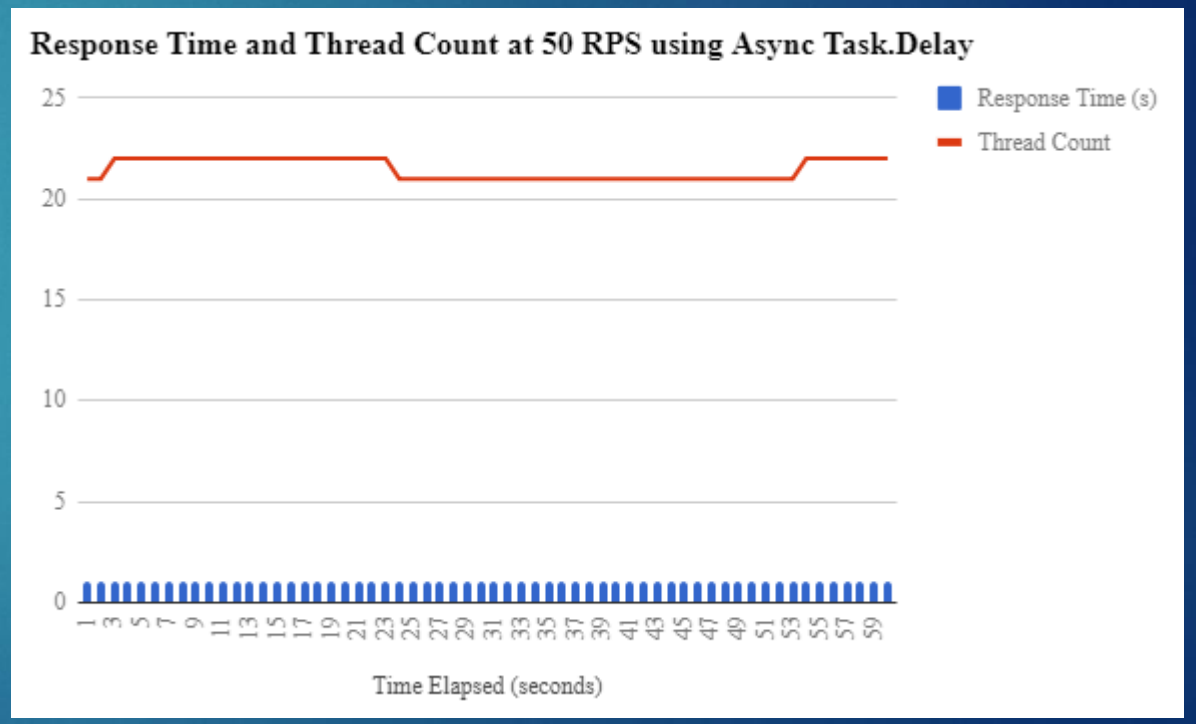
This is good! Using async/await

Scalability and performance in web applications

Asynchronous benefits

Improves throughput because of reduced number of threads usage, less memory use and less CPU use





Source: https://calebblloyd.com/software/net-core-mvc-thread-pool-vs-async/

Responsive user interfaces

Demo of a Web API running CPU-bound operation

- Targets .NET Core 3 Preview4 but works exactly the same with .NET Core 2
- The behaviour is the same in Web API using .NET 4.7.2 or earlier

Source code

https://github.com/gerardo-lijs/CSharpMeetup-Mar2019

Scalability and performance in web applications

Simple API method running an expensive CPU-bound operation

```
[Route("api/[controller]")]
[ApiController]
public class CPUBoundController : ControllerBase
{
    private int GetPrimesCount(int start, int count)
    {
        return ParallelEnumerable.Range(start, count).Count(n => Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0));
    }

    // GET api/cpubound
    // .\bombardier.exe "http://localhost:5000/api/cpubound?start=1&end=1000000" -n 20 -t 100s
    [HttpGet]
    public ActionResult<int> Get([FromQuery] int start, [FromQuery] int end)
    {
        return GetPrimesCount(start, end);
    }
}
```

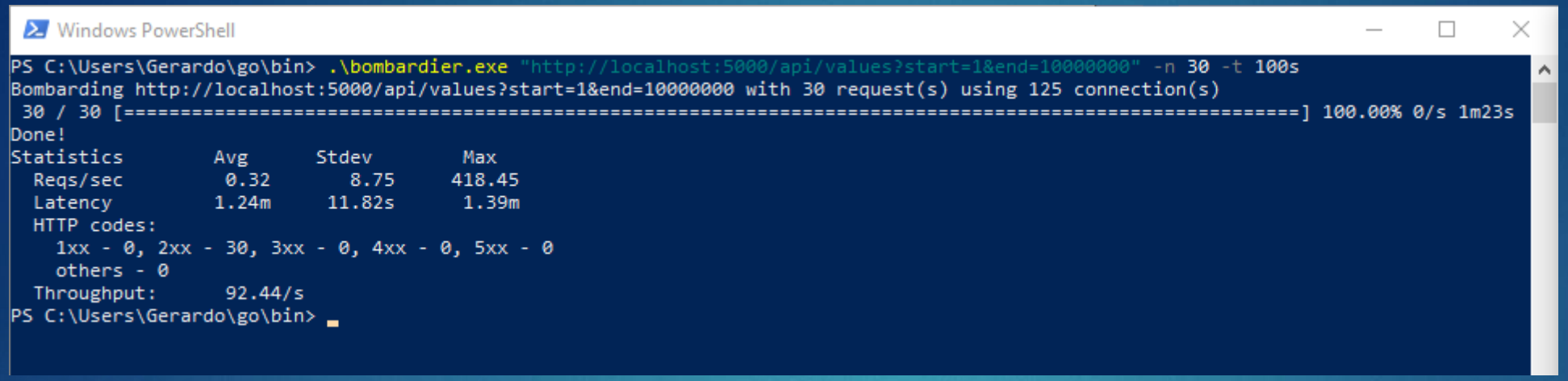
Scalability and performance in web applications

Same method refactored to use asynchronous code

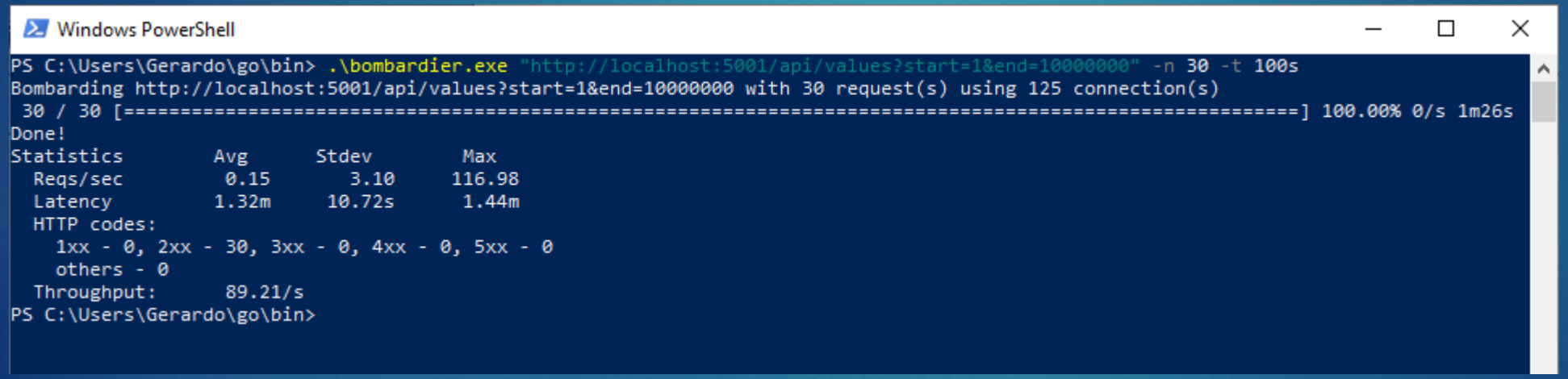
```
[Route("api/[controller]")]
[ApiController]
public class CPUBoundController : ControllerBase
{
    private Task<int> GetPrimesCountAsync(int start, int count)
    {
        return Task.Run(() =>
            ParallelEnumerable.Range(start, count).Count(n => Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0)));
    }

    // GET api/cpubound
    // .\bombardier.exe "http://localhost:5001/api/cpubound?start=1&end=1000000" -n 20 -t 100s
    [HttpGet]
    public async Task<ActionResult<int>> Get([FromQuery] int start, [FromQuery] int end)
    {
        return await GetPrimesCountAsync(start, end);
    }
}
```

Scalability and performance in web applications



Synchronous
CPU-bound



Asynchronous
CPU-bound

Using async/await makes it worse in this case!

Scalability and performance in web applications

Summary

- Use async with IO-bound operations whenever possible
- Don't block!
- Don't use async for expensive CPU-bound operations
- Use benchmarking tools such as Bombardier (https://github.com/codesenberg/bombardier)

I still need more performance, what can I do?

- Scale up
- Scale out whole application using Docker/Kubernetes
- Scale out only expensive CPU-bound operations to Azure Functions

Obsolete patterns

BackgroundWorker

```
backgroundWorker1.DoWork += new DoWorkEventHandler(backgroundWorker1_DoWork);
backgroundWorker1.RunWorkerCompleted += new RunWorkerCompletedEventHandler(backgroundWorker1_RunWorkerCompleted);
backgroundWorker1.ProgressChanged += new ProgressChangedEventHandler(backgroundWorker1_ProgressChanged);
```

Event-Based Asynchronous Pattern (EAP)

```
public void DownloadDataAsync(Uri address);
public event DownloadDataCompletedEventHandler DownloadDataCompleted;
public bool IsBusy { get; } // Indicates if still running
```

Asynchronous Programming Model (APM)

```
public IAsyncResult BeginRead(byte[] buffer, int offset, int size, AsyncCallback callback, object state);
public int EndRead(IAsyncResult asyncResult);
```

Task Cancellation

- Use CancellationToken in your method and check manually

```
async Task Foo(CancellationToken cancellationToken)
{
    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine(i);
        await Task.Delay(1000);
        cancellationToken.ThrowIfCancellationRequested();
    }
}
```

Task Cancellation

- Use Extension method

```
public static Task<TResult> WithCancellation<TResult>(this Task<TResult> task, CancellationToken cancelToken)
{
    var tcs = new TaskCompletionSource<TResult>();
    var reg = cancelToken.Register(() => tcs.TrySetCanceled());
    task.ContinueWith(ant =>
    {
        reg.Dispose();
        if (ant.IsCanceled)
            tcs.TrySetCanceled();
        else if (ant.IsFaulted)
            tcs.TrySetException(ant.Exception.InnerException);
        else
            tcs.TrySetResult(ant.Result);
    });
    return tcs.Task;
}
```

Run a Task with a Timeout

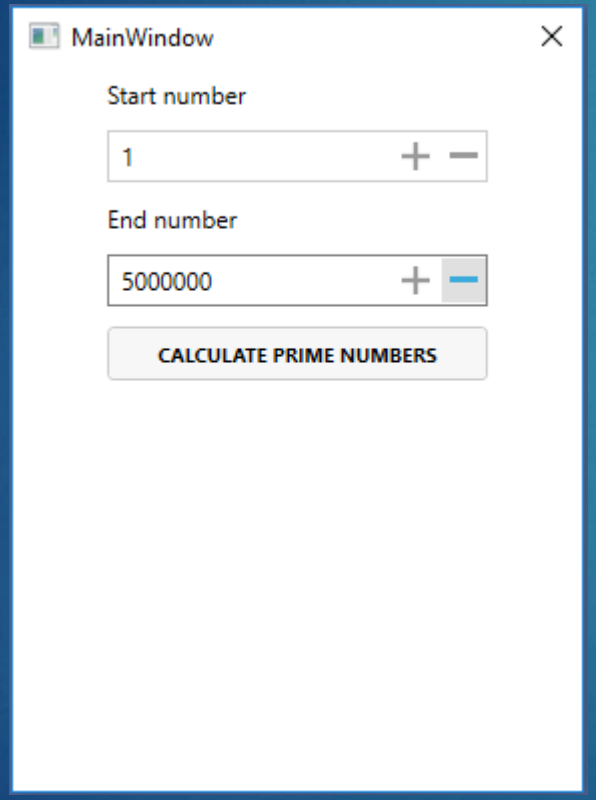
- Use Extension method

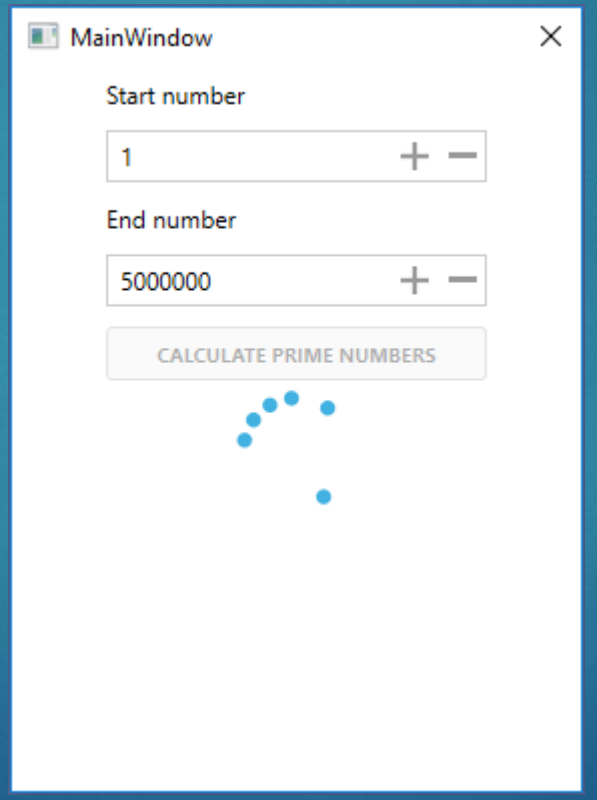
```
public async static Task<TResult> WithTimeout<TResult>(this Task<TResult> task, TimeSpan timeout)
{
    Task winner = await (Task.WhenAny(task, Task.Delay(timeout)));
    if (winner != task) throw new TimeoutException();
    return await task; // Unwrap result/re-throw
}
```

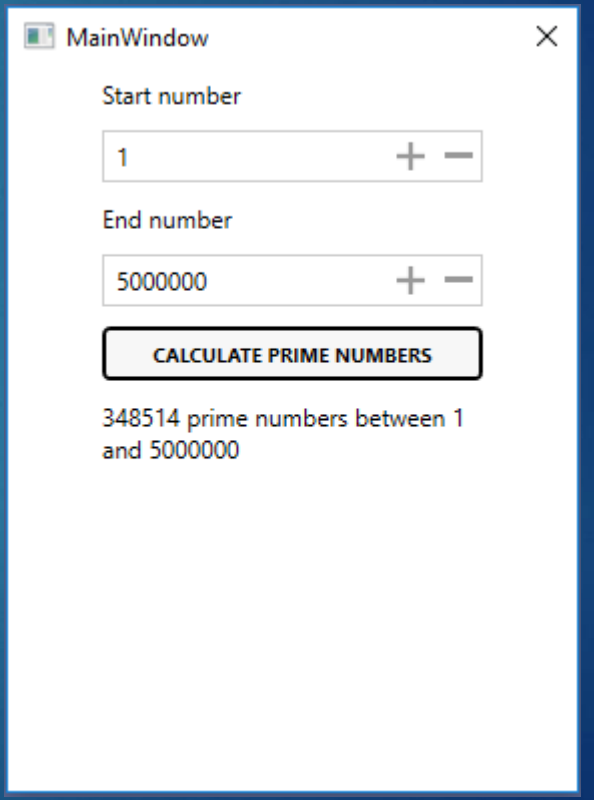
Source: David Fowler -> AspNetCoreDiagnosticScenarios/Scenarios/Infrastructure/TaskExtensions.cs

Progress reporting

- Use IProgress<T> interface
- Use indeterminate ProgressRing







Running asynchronous method synchronously

Use helper method borrowed from Microsoft

```
public static class AsyncHelper
{
    private static readonly TaskFactory _taskFactory = new
        TaskFactory(CancellationToken.None,
                    TaskCreationOptions.None,
                    TaskContinuationOptions.None,
                    TaskScheduler.Default);

    public static TResult RunSync<TResult>(Func<Task<TResult>> func)
        => _taskFactory
            .StartNew(func)
            .Unwrap()
            .GetAwaiter()
            .GetResult();

    public static void RunSync(Func<Task> func)
        => _taskFactory
            .StartNew(func)
            .Unwrap()
            .GetAwaiter()
            .GetResult();
}
```

Call like this

```
var result = AsyncHelper.RunSync(() => DoAsyncStuff());
```

Warning

Try to avoid this! Most of the times (probably always) there are better ways and you don't need to run async code synchronously

Source: https://cpratt.co/async-tips-tricks/

Async class constructors

They are not available by default in the framework

Option 1 - Refactor with InitializeAsync method

Previous class

```
public class ExampleClass
{
    public ExampleClass()
    {
        // Current initialization code here
    }
}
```

Refactored class

```
public class ExampleClass
{
    public ExampleClass()
    {
        // No more code here
    }

    public async Task InitializeAsync()
    {
        // Previous initialization code moved here
        // Plus you can call async methods with await now
        // Problem -> Consumers need to know they have to
        // always run this method after creating class
        // It's better to use Factory Pattern in my opinion
    }
}
```

Call like this

```
var classInstance = new ExampleClass();
await classInstance.InitializeAsync();
```

Async class constructors

They are not available by default in the framework

Option 2 - Refactor with Factory Pattern

Previous class

```
public class ExampleClass
{
    public ExampleClass()
    {
        // Current initialization code here
    }
}
```

Refactored class

```
public class ExampleClass
{
    // We explicitly create a private parameterless constructor
    // so that we are forced to use the static async method
    private ExampleClass() { }

    private async Task InitializeAsync()
    {
        // Previous initialization code moved here
        // Plus you can call async methods with await now
    }

    public static async Task<ExampleClass> CreateAsync()
    {
        var ret = new ExampleClass();
        await ret.InitializeAsync();
        return ret;
    }
}
```

Call like this

```
var classInstance = await ExampleClass.CreateAsync();
```

Parallel programming

Demo of a WPF application using TPL and Task.WhenAll

- Uses ReactiveUI and MahApps but they are not a requirement
- Uses OpenCvSharp library as example of expensive operations
- Targets .NET Core 3 Preview4 but works exactly the same with .NET Core 2
- The behaviour is the same in Web API using .NET 4.7.2 or earlier

Source code

https://github.com/gerardo-lijs/CSharpMeetup-Mar2019

Slides, resources and source code

https://github.com/gerardo-lijs/CSharpMeetup-Mar2019

https://github.com/gerardo-lijs/Asynchronous-Programming

Gerardo Lijs
Software Developer At Microptic SL
Email gerardo.lijs@gmail.com
Twitter @GerardoLijs

Thank You!