10 YEARS

07.NOV.2024

ELEVATING THE DEVELOPERS' COMMUNITY

devday
GET INSPIRED

PLATINIUM

ONIRYX

GOLD

delaware

GAMING1

GENESIS CONSULT
IT Consultancy Services

PARTNER

birdit

DevApps.be

meet innovate create

SENSE OF TECH

Sparkle

devday GET INSPIRED

Async Await tips & tricks

Sébastien Pertus

Microsoft ISE (Industry Solutions Engineering)

@sebpertus

devday GET INSPIRED

David Fowler Guidance

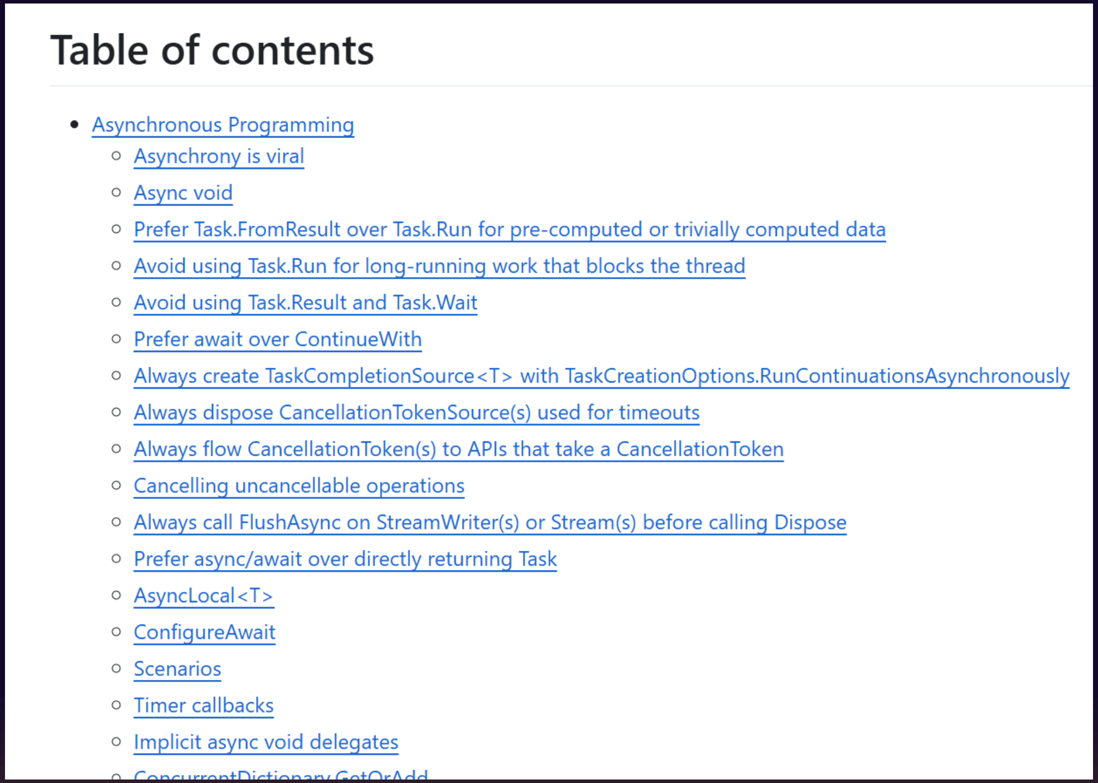


Table of contents

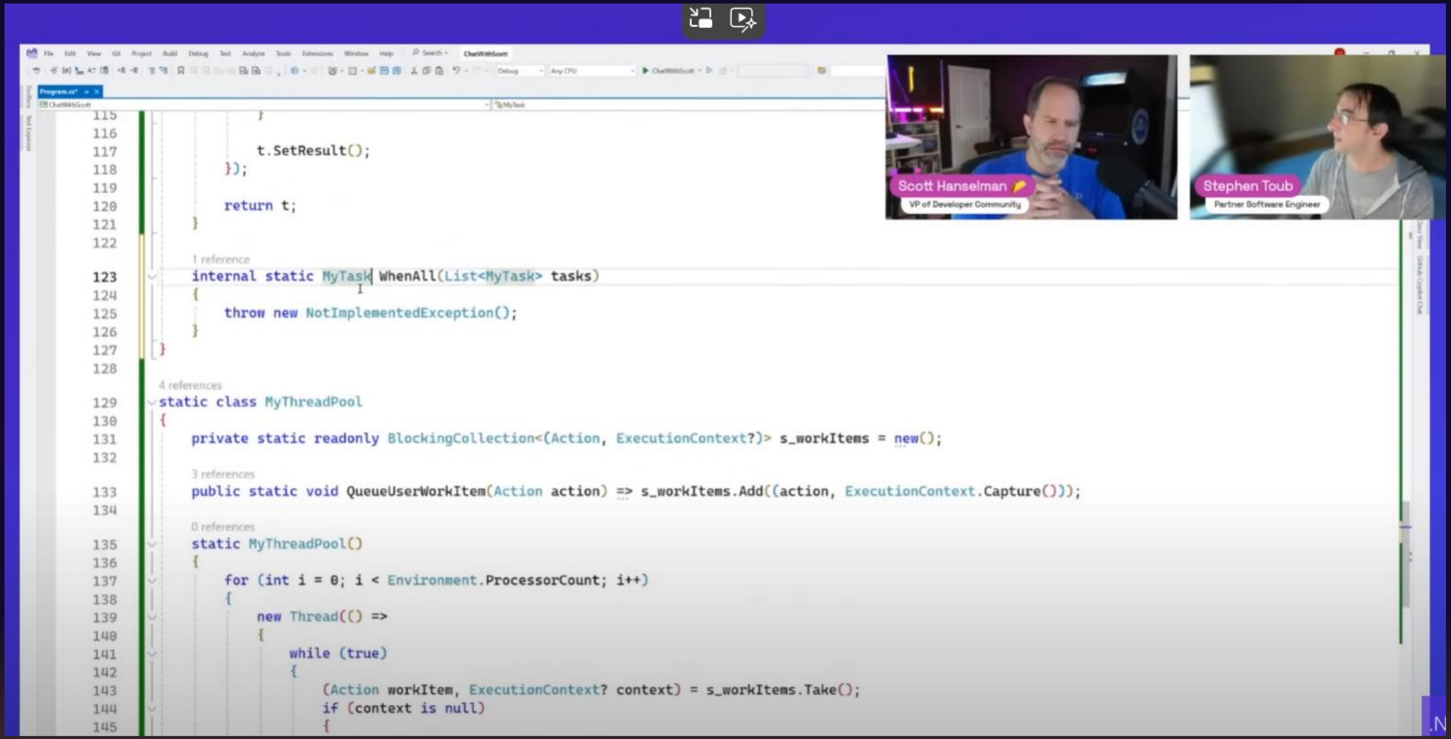
- Asynchronous Programming
  - Asynchrony is viral
  - Async void
  - Prefer Task.FromResult over Task.Run for pre-computed or trivially computed data
  - Avoid using Task.Run for long-running work that blocks the thread
  - Avoid using Task.Result and Task.Wait
  - Prefer await over ContinueWith
  - Always create TaskCompletionSource<T> with TaskCreationOptions.RunContinuationsAsynchronously
  - Always dispose CancellationTokenSource(s) used for timeouts
  - Always flow CancellationToken(s) to APIs that take a CancellationToken
  - Cancelling uncancellable operations
  - Always call FlushAsync on StreamWriter(s) or Stream(s) before calling Dispose
  - Prefer async/await over directly returning Task
  - AsyncLocal<T>
  - ConfigureAwait
  - Scenarios
  - Timer callbacks
  - Implicit async void delegates
  - ConcurrentDictionary.GetOrAdd

- AspNetCoreDiagnosticScenarios/AsyncGuidance.md at master · davidfowl/AspNetCoreDiagnosticScenarios

devday GET INSPIRED

Stephen Toub video on Task

- Deep .NET: Writing async/await from scratch in C# with Stephen Toub and Scott Hanselman



```
            }

            t.SetResult();
        });

        return t;
    }

    1 reference
    internal static MyTask WhenAll(List<MyTask> tasks)
    {
        throw new NotImplementedException();
    }
}

4 references
static class MyThreadPool
{
    private static readonly BlockingCollection<(Action, ExecutionContext?)> s_workItems = new();

    3 references
    public static void QueueUserWorkItem(Action action) => s_workItems.Add((action, ExecutionContext.Capture()));

    0 references
    static MyThreadPool()
    {
        for (int i = 0; i < Environment.ProcessorCount; i++)
        {
            new Thread(() =>
            {
                while (true)
                {
                    (Action workItem, ExecutionContext? context) = s_workItems.Take();
                    if (context is null)
                    {
```

Disclaimer

- You know async await...
- But let’s make a quick demo ☺

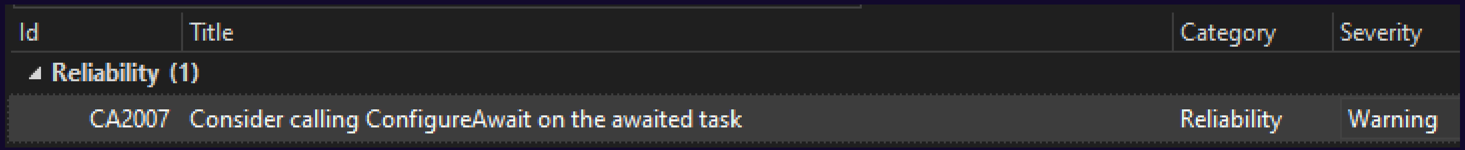
Task / async / await

- Asynchronous vs multi threading

- “On peut avoir de l’asynchronisme sans multi threading”
- “On ne peut pas avoir de multi threading sans asynchronisme”

ConfigureAwait(false)

- Use ConfigureAwait(false)
  - Except: When need to return to calling thread
  - Only if your framework is using a SynchronizationContext mechanism
    - WPF / Winforms / Xamarin / .NET MAUI / WinUI / Blazor
    - ASP.Net (> Core) IS NOT using any SynchronizationContext



Id	Title	Category	Severity
◢ Reliability (1)			
CA2007	Consider calling ConfigureAwait on the awaited task	Reliability	Warning

ConfigureAwait(ConfigureAwaitOptions)

- ConfigureAwait(ConfigureAwaitOptions options)
  - ContinueOnCapturedContext: Attempt to return to calling thread
    - Like ConfigureAwait(true)
  - ForceYielding: Forces an await on an already completed Task
    - Like the task was not yet completed
  - SuppressThrowing: Suppress any exception on faulted task
  - None: Will not return to the calling thread
    - Like ConfigureAwait(false)

- Enums options can be combined

ConfigureAwait

- Use it in library
- Don’t use it in UI application & ASP.NET

- Add ConfigureAwait(false) by roji · Pull Request #21110 · dotnet/efcore
- Add missing .ConfigureAwait(false) in HttpConnectionPool by stephentoub · Pull Request #38610 · dotnet/corefx

01-Demo

.Wait() / .Result . / GetAwaiter().GetResult()

- Don’t use .Wait() or .Result()
  - Always use await if you can

- In a Synchronous world, from an async world ...
  - .Wait() or .Result(): Wraps all exceptions in an AggregateException
  - .GetAwaiter().GetResult(): Propagates exceptions correctly

- **HOWEVER**, any solution will probably cause a **deadlock** somehow...
- **Try to avoid to run any async task in a sync way...**

.Wait();

```
public void ExceptionHandlingUsingWaitMethod()
{
   var task = AnTaskAsync();
   try
   {
      task.Wait();
   }
   catch (AggregateException e)
   {
      foreach (var innerException in e.InnerExceptions)
      {
         Console.WriteLine(innerException.Message);
         throw;
      }
   }
}
```

.GetAwaiter().GetResult()

```
public void ExceptionHandlingUsingGetAwaiterMethod()
{
   var task = AnTaskAsync();
   try
   {
      task.GetAwaiter().GetResult();
   }
   catch (InvalidOperationException e)
   {
      Console.WriteLine($"Error Message: {e.Message}");
      throw;
   }
}
```

devday
GET INSPIRED

async void

- Don’t use async void
  - Task holds the exception, if any, where async void does not since there is no returning Task .
- Void-returning async methods have a specific purpose: to make asynchronous event handlers possible (ref)

- Void-returning method can potentially allow us to fire a method and forget about it:
  - Task.FireAndForget()

02-Demo

devday
GET INSPIRED

And Here
Comes
the drama
quickmeme.com

Returning task directly

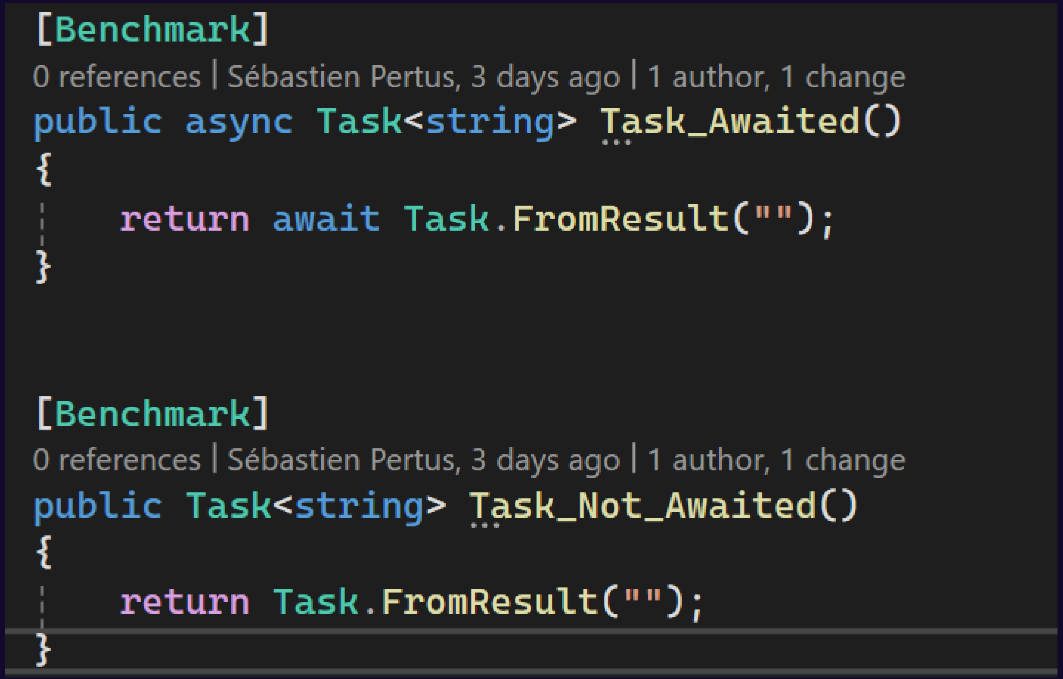
- Avoid “return async”
  - Or not …
- Return directly your task, except:
  - try / catch
  - When used with using blocks

```
public Task<string> GetOrdersAsync ()
{
   return File.ReadAllTextAsync("orders1.json");
}

public async Task<string> GetOrders2Async()
{
   return await File.ReadAllTextAsync("orders2.json");
}
```

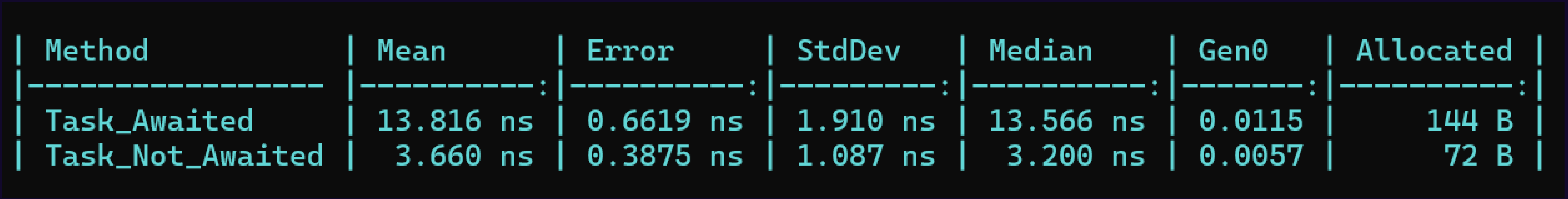
03-Demo

devday GET INSPIRED

Returning tasks directly

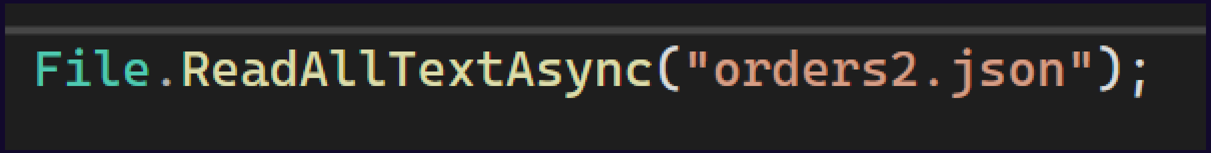


```
[Benchmark]
0 references | Sébastien Pertus, 3 days ago | 1 author, 1 change
public async Task<string> Task_Awaited()
{
    return await Task.FromResult("");
}


[Benchmark]
0 references | Sébastien Pertus, 3 days ago | 1 author, 1 change
public Task<string> Task_Not_Awaited()
{
    return Task.FromResult("");
}
```

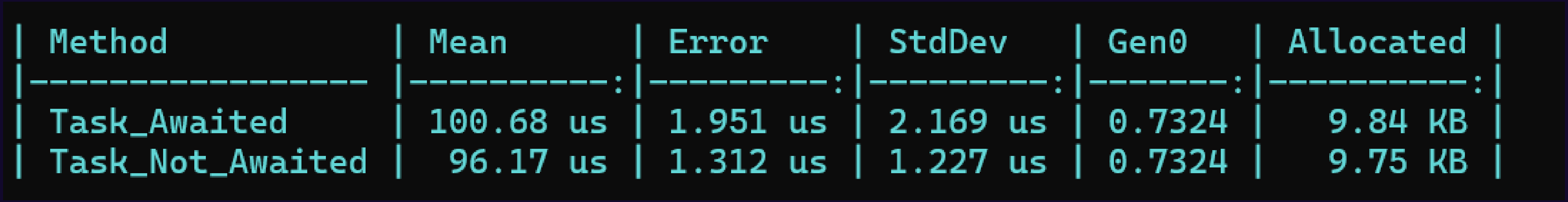


```
| Method           | Mean      | Error     | StdDev    | Median    | Gen0   | Allocated |
|----------------- |----------:|----------:|----------:|----------:|-------:|----------:|
| Task_Awaited     | 13.816 ns | 0.6619 ns | 1.910 ns  | 13.566 ns | 0.0115 |     144 B |
| Task_Not_Awaited |  3.660 ns | 0.3875 ns | 1.087 ns  |  3.200 ns | 0.0057 |      72 B |
```

Returning tasks directly



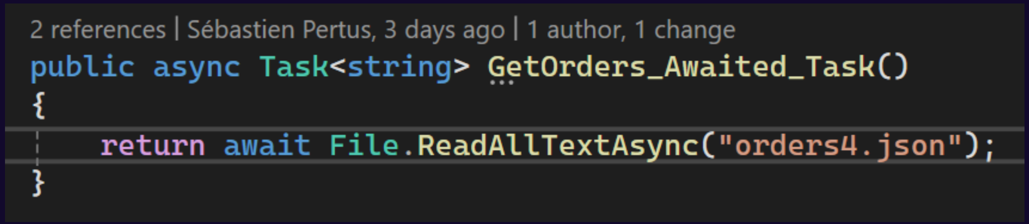
```
File.ReadAllTextAsync("orders2.json");
```



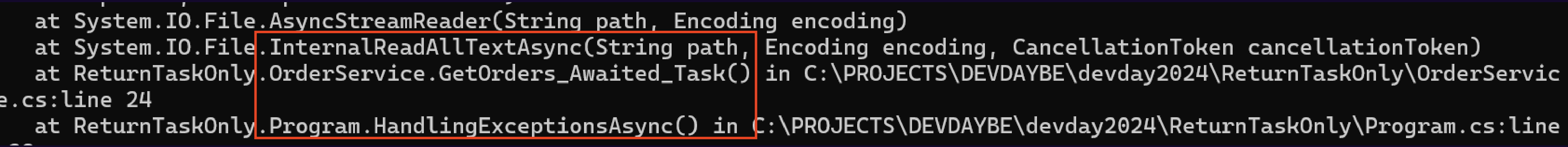
```
| Method            | Mean      | Error    | StdDev   | Gen0   | Allocated |
|------------------ |----------:|---------:|---------:|-------:|----------:|
| Task_Awaited      | 100.68 us | 1.951 us | 2.169 us | 0.7324 |   9.84 KB |
| Task_Not_Awaited  |  96.17 us | 1.312 us | 1.227 us | 0.7324 |   9.75 KB |
```

devday GET INSPIRED

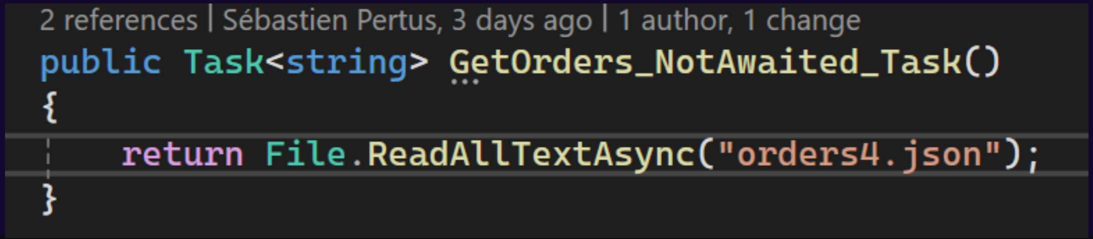
Returning tasks directly



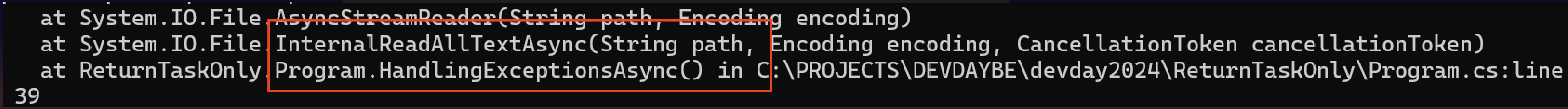
```
2 references | Sébastien Pertus, 3 days ago | 1 author, 1 change
public async Task<string> GetOrders_Awaited_Task()
{
    return await File.ReadAllTextAsync("orders4.json");
}
```



```
   at System.IO.File.AsyncStreamReader(String path, Encoding encoding)
   at System.IO.File.InternalReadAllTextAsync(String path, Encoding encoding, CancellationToken cancellationToken)
   at ReturnTaskOnly.OrderService.GetOrders_Awaited_Task() in C:\PROJECTS\DEVDAYBE\devday2024\ReturnTaskOnly\OrderServic
e.cs:line 24
   at ReturnTaskOnly.Program.HandlingExceptionsAsync() in C:\PROJECTS\DEVDAYBE\devday2024\ReturnTaskOnly\Program.cs:line
```



```
2 references | Sébastien Pertus, 3 days ago | 1 author, 1 change
public Task<string> GetOrders_NotAwaited_Task()
{
    return File.ReadAllTextAsync("orders4.json");
}
```



```
   at System.IO.File.AsyncStreamReader(String path, Encoding encoding)
   at System.IO.File.InternalReadAllTextAsync(String path, Encoding encoding, CancellationToken cancellationToken)
   at ReturnTaskOnly.Program.HandlingExceptionsAsync() in C:\PROJECTS\DEVDAYBE\devday2024\ReturnTaskOnly\Program.cs:line
 39
```

04-Demo

devday GET INSPIRED

TaskCompletionSource

- Useful to encapsulate legacy EAP / APM

```
var tcs = new TaskCompletionSource<string>(TaskCreationOptions.RunContinuationsAsynchronously);

tcs.SetResult(token);
tcs.SetException(ex);

var result = await tcs.Task;
```

Asynchronous models

- Task-based Asynchronous Pattern (**TAP**) (old, don’t use it)
- Event-based Asynchronous Pattern (**EAP**) (old, don’t use it)
- Asynchronous Programming Model (**APM**) (old…ish, use it !)

```
public class WebAuthenticationTAP
{
   public async Task<string> AuthenticateAsync(string username, string password) { }
}

public class WebAuthenticationEAP
{
   public void AuthenticateAsync(string username, string password) { }
   public event Action<string> AuthenticationCompleted;
   public event Action<Exception> AuthenticationFailed;
}

public class WebAuthenticationAPM
{
   public IAsyncResult BeginAuthenticate(string name, string pwd, AsyncCallback callback, object state) { }
   public string EndAuthenticate(IAsyncResult result) { }
}
```

From APM to TAP

```
return Task<string>.Factory.FromAsync(
                    webLegacyAuthentication.BeginAuthenticate,
              webLegacyAuthentication.EndAuthenticate,
                    username, password, null);
```

05-Demo

devday GET INSPIRED

ValueTask

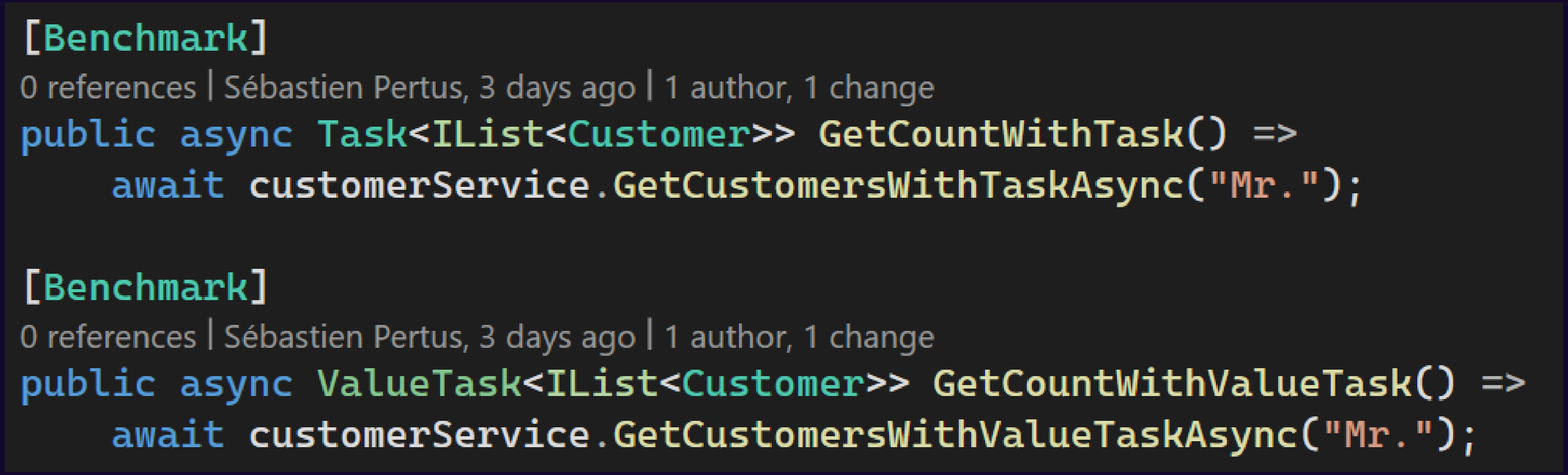
- ValueTask<TResult> was introduced in .NET Core 2.0

- Using ValueTask when
  - Your method will not “*most of the time*” go through an await call
    - Example: When caching a value from your database
    - Example: When caching a value read from disk

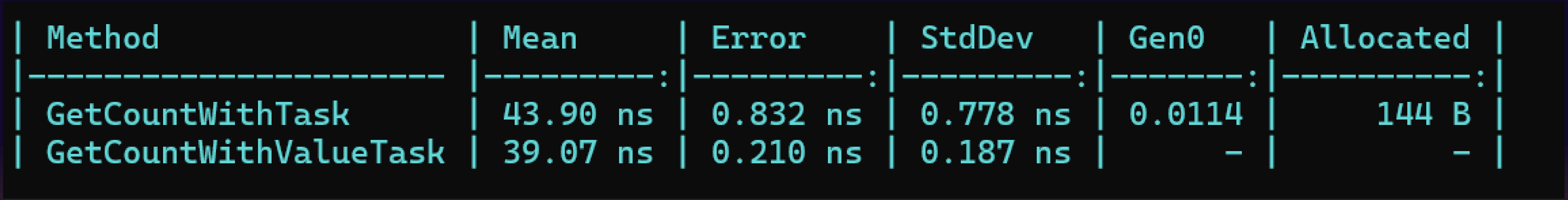
06-Demo

devday GET INSPIRED

ValueTask



```
[Benchmark]
0 references | Sébastien Pertus, 3 days ago | 1 author, 1 change
public async Task<IList<Customer>> GetCountWithTask() =>
    await customerService.GetCustomersWithTaskAsync("Mr.");

[Benchmark]
0 references | Sébastien Pertus, 3 days ago | 1 author, 1 change
public async ValueTask<IList<Customer>> GetCountWithValueTask() =>
    await customerService.GetCustomersWithValueTaskAsync("Mr.");
```



```
| Method                | Mean     | Error    | StdDev   | Gen0   | Allocated |
|---------------------- |---------:|---------:|---------:|-------:|----------:|
| GetCountWithTask      | 43.90 ns | 0.832 ns | 0.778 ns | 0.0114 |     144 B |
| GetCountWithValueTask | 39.07 ns | 0.210 ns | 0.187 ns |      - |         - |
```

ValueTask: Should we replace every Task ?

- No... Use “by default” Task
- Task is easier to use and will ensure all scenarios
  - Most of the time, performances are enough
- Minor costs with ValueTask<TResult> instead of a Task<TResult>
  - Microbenchmarks it’s a bit faster to await a Task<TResult> vs ValueTask<TResult>,

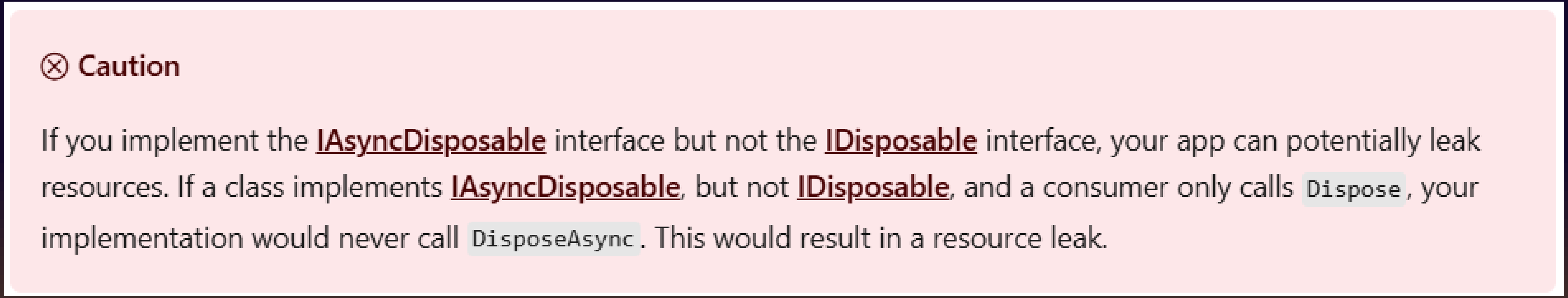
ValueTask: When avoid using ValueTask

```
// Given this ValueTask<int>-returning method…
public ValueTask<int> SomeValueTaskReturningMethodAsync();
// GOOD
int result = await SomeValueTaskReturningMethodAsync();
int result = await SomeValueTaskReturningMethodAsync().ConfigureAwait(false);
Task<int> t = SomeValueTaskReturningMethodAsync().AsTask();

// WARNING
ValueTask<int> vt = SomeValueTaskReturningMethodAsync();
// storing the instance into a local makes it much more likely it'll be misused,

// BAD: awaits multiple times
ValueTask<int> vt = SomeValueTaskReturningMethodAsync();
int result = await vt;
int result2 = await vt;

// BAD: awaits concurrently (and, by definition then, multiple times)
ValueTask<int> vt = SomeValueTaskReturningMethodAsync();
Task.Run(async () => await vt);
Task.Run(async () => await vt);

// BAD: uses GetAwaiter().GetResult() when it's not known to be done
ValueTask<int> vt = SomeValueTaskReturningMethodAsync();
int result = vt.GetAwaiter().GetResult();
```

IAsyncDisposable

- Implemented in C#~8
- Allows you to create a disposable async object
- A lot of existing object are already using IAsyncDisposable

```
await using (var fileStreamText = new FileStream("file.txt", FileMode.Create))
{
    // Do something with the fileStream
}
```



ⓧ **Caution**

If you implement the IAsyncDisposable interface but not the IDisposable interface, your app can potentially leak resources. If a class implements IAsyncDisposable, but not IDisposable, and a consumer only calls `Dispose`, your implementation would never call `DisposeAsync`. This would result in a resource leak.

07-Demo

devday GET INSPIRED

IAsyncEnumerable

- For streaming data
  - Allow us to update the UI during the process
  - Use [EnumeratorCancellation] for CancellationToken

```
await foreach (var customer in CustomerService.GetALotOfCustomersByPageAsync())
{
   Console.WriteLine(customer);
}
```

08-Demo

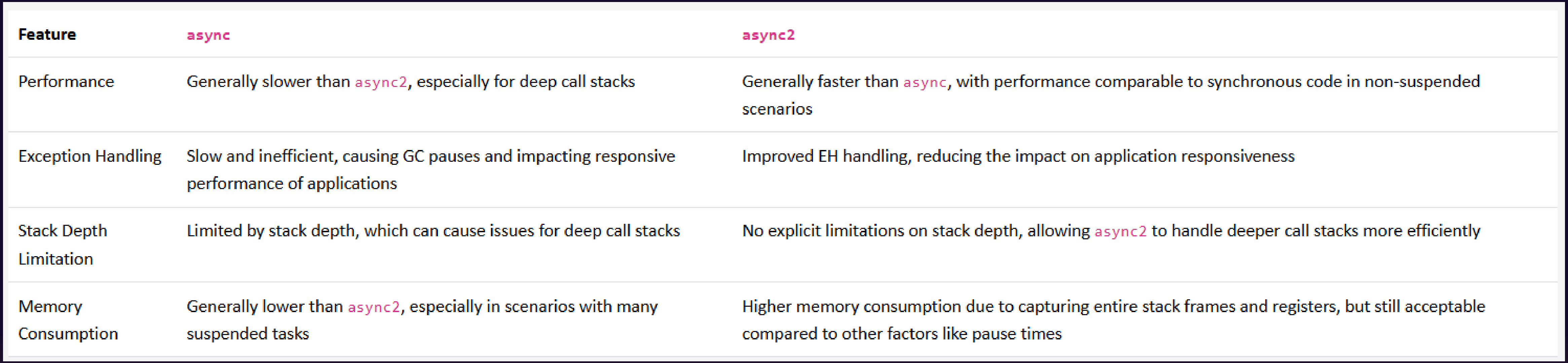
WaitAsync(Timeout, CancellationToken)

- Append to any async method lacking a cancellation token,
- Allow to cancel any method using a timeout,

```
public class Task
{
    public Task WaitAsync(CancellationToken cancellationToken) => WaitAsync(Timeout.UnsignedInfinite, cancellationToken);

    public Task WaitAsync(TimeSpan timeout) => WaitAsync(ValidateTimeout(timeout, ExceptionArgument.timeout), default);

    public Task WaitAsync(TimeSpan timeout, CancellationToken cancellationToken) =>
                                                WaitAsync(ValidateTimeout(timeout, ExceptionArgument.timeout), cancellationToken);
}
```

Async2 ?

[runtimelab/docs/design/features/runtime-handled-tasks.md at feature/async2-experiment · dotnet/runtimelab]()



Feature	`async`	`async2`
Performance	Generally slower than `async2`, especially for deep call stacks	Generally faster than `async`, with performance comparable to synchronous code in non-suspended scenarios
Exception Handling	Slow and inefficient, causing GC pauses and impacting responsive performance of applications	Improved EH handling, reducing the impact on application responsiveness
Stack Depth Limitation	Limited by stack depth, which can cause issues for deep call stacks	No explicit limitations on stack depth, allowing `async2` to handle deeper call stacks more efficiently
Memory Consumption	Generally lower than `async2`, especially in scenarios with many suspended tasks	Higher memory consumption due to capturing entire stack frames and registers, but still acceptable compared to other factors like pause times

Async guidance by David Fowler

AspNetCoreDiagnosticScenarios/AsyncGuidance.md at master · davidfowl/AspNetCoreDiagnosticScenarios