

# Смысл асинхронности, ошибки

## в ASP.NET Core

# Что это?

- **Асинхронное программирование** – разновидность конкурентности, использующая обещания или обратные вызовы **для предотвращения** создания лишних потоков
- **Конкурентность** – выполнение сразу нескольких действий в одно и то же время

## Простыми словами

- Синхронный вызов: когда мы вызываем метод, то мы получим управления только после завершения этой операции
- Асинхронный: ОК, я начал операцию, сразу возвращаю тебе управления, и каким-то образом потом сообщу тебе о завершении операции
- Асинхронность **не требует** создания нового потока

# Зачем нужна асинхронность?

3 причины:

1. Большинство UI фреймворков работают на одном потоке, а блокировка этого потока равнозначна блокировке всего UI
2. Каждый поток (даже ожидающий) потребляет ресурсы ОС
3. [ThreadPool starvation](#) (об этом позже)

## Ключевые слова `async/await`

- Ключевое слово `await` **асинхронно** меняет `Task<T>` на результат `T`
- Метод, помеченный модификатором `async` позволяет использовать `await` в коде такого метода, а также разворачивает машину состояний (конечный автомат)
- `async` можно применять к `Task`, `Task<T>`, `ValueTask<T>`, `IAsyncEnumerable<T>` и к `void` (но с `void` лучше не надо)

# Объявление асинхронного метода

```
async Task Wait1000msAsync()
```

```
{
```

```
    await Task.Delay(1000);
```

```
    //управления вернется сюда после ожидания 1000 мс
```

```
    //при этом поток не будет заблокирован
```

```
}
```

## Объявление асинхронного метода

```
async Task<string> ReadMyFileAsync()  
{  
    var content = await File.ReadAllTextAsync("file.txt");  
    return content.Replace("\t", " ");  
}
```

# Магия async/await: Машина состояний

```
Code C# Create Gist Default Results C# Debug
using System;
using System.Threading.Tasks;

public class C {
    async Task Wait1000msAsync()
    {
        await Task.Delay(1000);
    }
}

[CompilerGenerated]
private sealed class <Wait1000msAsync>d__0 : IAsyncStateMachine
{
    public int <>1__state;

    public AsyncTaskMethodBuilder <>t__builder;

    public C <>4__this;

    private TaskAwaiter <>u__1;

    private void MoveNext()
    {
        int num = <>1__state;
        try
        {
            TaskAwaiter awaiter;
            if (num != 0)
            {
                awaiter = Task.Delay(1000).GetAwaiter();
                if (!awaiter.IsCompleted)
                {
                    num = (<>1__state = 0);
                    <>u__1 = awaiter;
                    <Wait1000msAsync>d__0 stateMachine = this;
                    <>t__builder.AwaitUnsafeOnCompleted(ref awaiter, ref stateMachine, this);
                    return;
                }
            }
            else
            {
                awaiter = <>u__1;
                <>u__1 = default(TaskAwaiter);
                num = (<>1__state = -1);
            }
            awaiter.GetResult();
        }
        catch (Exception exception)
        {
            <>1__state = -2;
            <>t__builder.SetException(exception);
            return;
        }
        <>1__state = -2;
    }
}
```

Editor: Default Theme: Auto Built by Andrey Shchekin (@ashmind) – see SharpLab on GitHub.



# Подробнее про async/await и машину состояний

Yield и async-await: как оно все устроено внутри и как этим воспользоваться – Иван Дашкевич

CLRium #6: async/await. Машина состояний (Дмитрий Тихонов)

Dissecting the async methods in C# – Sergey Tepliakov

# Асинхронность

В ASP.NET Core

# Асинхронность

- Блокирующие вызовы блокируют и потоки
- А каждый поток потребляет ресурсы ОС
- Асинхронные же вызовы отпускают потоки, тем самым потребляя меньше ресурсов
- В итоге сервер, на котором используются асинхронные методы сможет обработать больше запросов, чем сервер с блокирующими методами

# Асинхронность

`Thread.Sleep(1000)` vs `await Task.Delay(1000);` ✓

# Асинхронность

```
var text = File.ReadAllText("file.txt");
```

vs

```
var text = await File.ReadAllTextAsync("file.txt");
```



# Асинхронность

```
var orders = Set<Order>().Where(o => o.Price > 100).ToList();
```

VS

```
var orders = await Set<Order>().Where(o => o.Price > 100).ToListAsync();
```



# Асинхронность vs Многопоточность

- Асинхронность используется в основном для IO операций
  - Работа с сетью, в т. ч. с БД
  - Работа с файловой системой
  - Задержки (Delay)
- Многопоточность для CPU bound операций
  - Тяжелые вычисления
  - Рендеринг

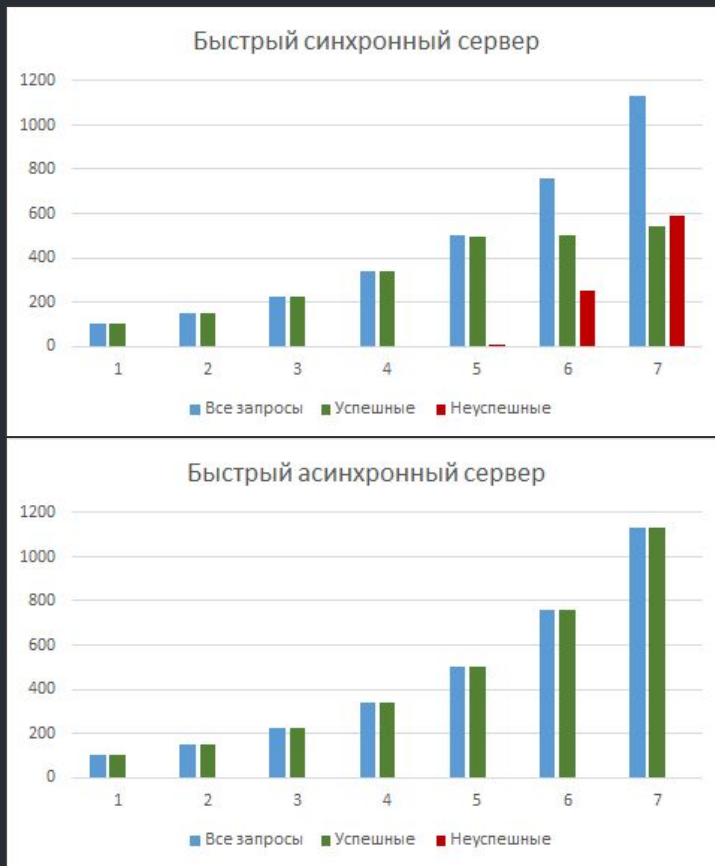
## Как работает честная асинхронность?

- Честные асинхронные вызовы работают без создания НОВЫХ ПОТОКОВ
- В этом случае работа делегируется другим устройствам:
  - Сетевые запросы делегируются сетевой карте
  - Запросы к файловой системе делегируются жесткому диску
  - Запрос на задержку (счетчик) обычно делегируется процессорному счетчику



# Sync vs Async

- Синхронный сервер
  - При нагрузке в 1100 RPS успешно выполнены только 500 запросов
- Асинхронный сервер
  - При нагрузке в 1100 RPS все запросы успешно выполнены
- [Исходники](#) (© Марк Шевченко)



# Sync over Async

- `task.Wait()`, `task.Result`, `task.GetAwaiter().GetResult()`
- При вызове асинхронного метода как синхронного в лучшем случае вы лишитесь преимуществ асинхронности, т. к. вызывающий поток заблокируется на время вызова асинхронного метода
- На самом деле использование sync over async получается даже дороже, чем вызов честных блокирующих методов
- Также провоцирует ThreadPool starvation
- В среде выполнения с контекстом синхронизации (WinForms, WPF, Unity) вся ваша программа войдет в дедлок и заблокируется навсегда (попросту-говоря зависнет)

# Sync over Async в WPF и т. д. = Deadlock

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    WaitOneSec().GetAwaiter().GetResult(); //дедлок случится здесь
    MessageBox.Show("Это сообщение не покажется никогда");
}
```

```
private async Task WaitOneSec()
    ⇒ await Task.Delay(TimeSpan.FromSeconds(1));
```

## SynchronizationContext (контекст синхронизации)

- SynchronizationContext выступает в роли интерфейса доступа к некоторым потокам. Как IEnumerable<T>, отдав кому-либо SynchronizationContext вы не сообщаете принимающей стороне подробности реализации. Принимающая сторона в зависимости от того, какой конкретно контексты передали будет планировать исполнение кода либо на ThreadPool либо на потоке UI, либо где-то ещё.
- Актуален в WPF, WinForms, старый ASP.NET

## ThreadPool starvation (истощение пула потоков)

- Происходит, когда нет свободных потоков для обработки поставленных в очередь тасок
- Возникает при использовании блокирующих вызовов в потоках ThreadPool'a
- При этом, CLR отвечает увеличением числа потоков ThreadPool
- Диагностика threadpool starvation при помощи утилиты counters

# ThreadPool starvation: lock

проверить

# ThreadPool starvation

DEMO

## ValueTask<T>

- Легковесные задачи, т. к. не выделяют память в куче
- Используется как возвращаемый тип в ситуациях, в которых обычно может быть возвращен синхронный результат, а асинхронное поведение встречается реже
- Используется там, где перфоманс **очень** важен
- Но накладывает кучу ограничений, о них далее



## ValueTask<T>: Ограничения

- Можно потребить только 1 раз, поэтому вызывается **обязательно с await**
- Нельзя вызывать блокирующую версию: `GetAwaiter().GetResult()` или `Result` или `Wait()`
- Нельзя использовать в методах `Task.WhenAll`, `Task.WaitAll`
- Если все-таки хочется использовать с `WhenAll`, то воспользуйтесь методом `AsTask()`
- Большинство методов должно возвращать `Task<T>`, поскольку при потреблении `Task<T>` возникает меньше скрытых ловушек, чем при потреблении `ValueTask<T>`

# ValueTask<T>: Пример

# ConfigureAwait

```
await Task.Delay(1000).ConfigureAwait(continueOnCapturedContext: false);
```

- `false` означает, что выполнение НЕ будет продолжено в потоке из контекста синхронизации. Т. е. в случае вызова из UI, выполнение продолжится НЕ в UI потоке, а в рабочем потоке из тредпула.
- Применяется в основном в библиотеках
- **В ASP.NET Core использовать не следует, т. к.**

# IAsyncDisposable

Паттерн высвобождения ресурсов, позволяющий в методе `DisposeAsync` использовать асинхронные вызовы

# Ошибки асинхронного программирования

## async void

- Невозможно использовать `await` (т.е. ожидать результат)
- Невозможно определить когда выполнение завершится
- Вызывающий код не может перехватить возникающие исключения
- Не следует применять примерно нигде

## Задача: Это асинхронный код или нет?

```
var result = await Task.Run(() => mailSender.Send(...));
```

- Да, но это нечестная асинхронность
- Такой код имеет смысл в исполняющей среде с контекстом синхронизации (UI, например)

# Дополнительные материалы

[AsyncGuidance](#)