# 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)

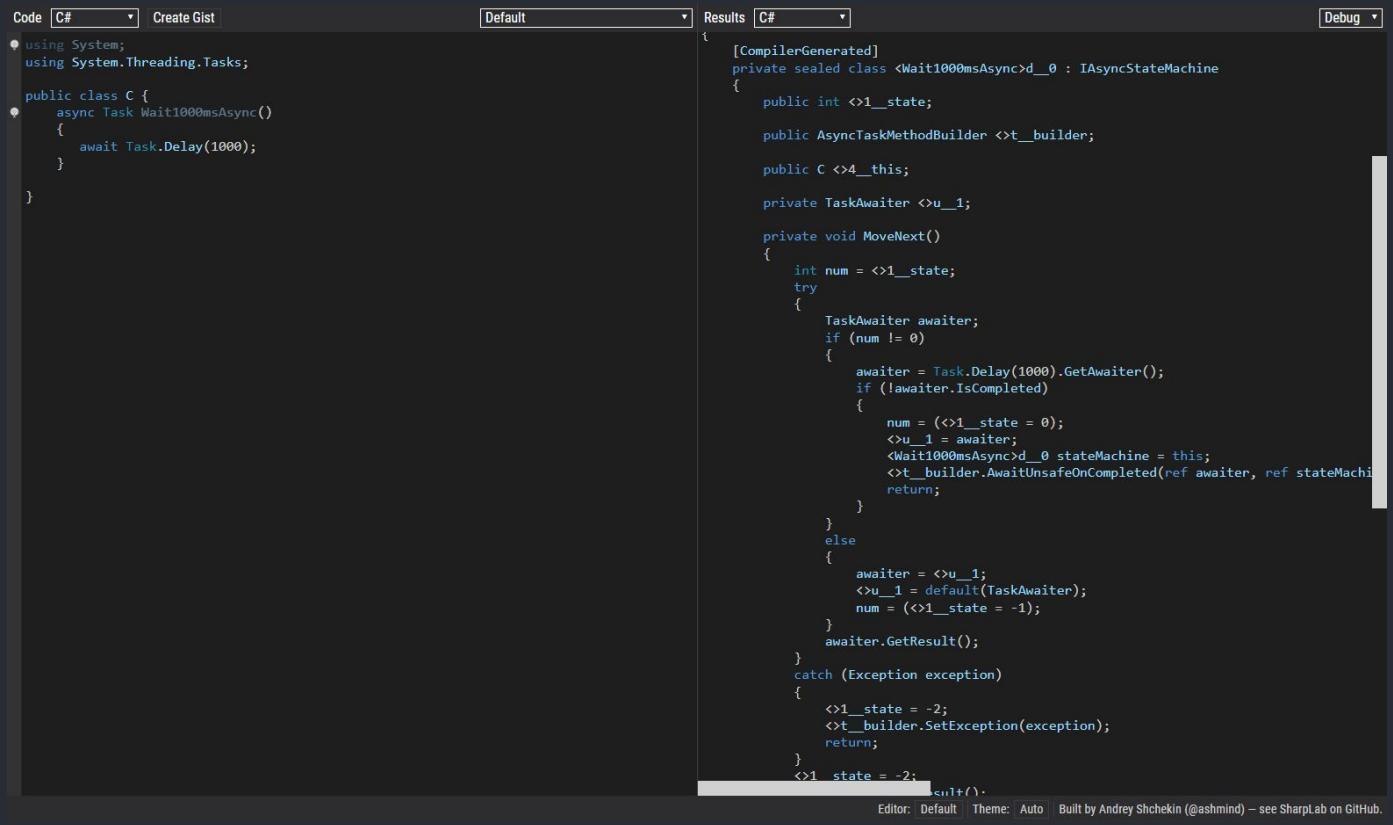
# Объявление асинхронного метода

```
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



# 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 сможем 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

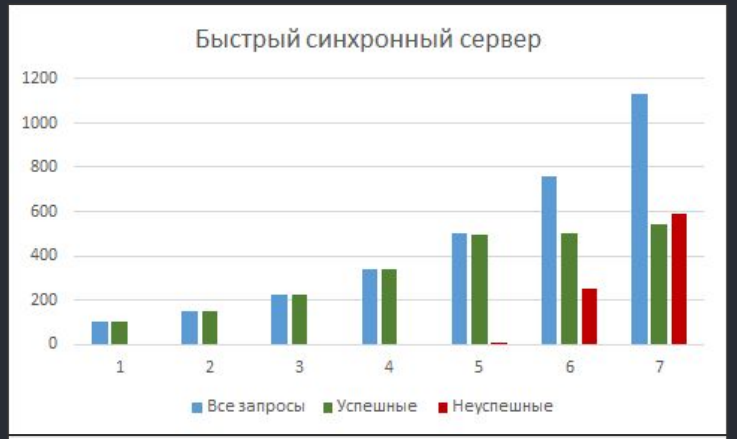
- Асинхронность используется в основном для IO операций
  - Работа с сетью, в т. ч. с БД
  - Работа с файловой системой
  - Задержки (Delay)
- Многопоточность для CPU bound операций
  - Тяжелые вычисления
  - Рендеринг

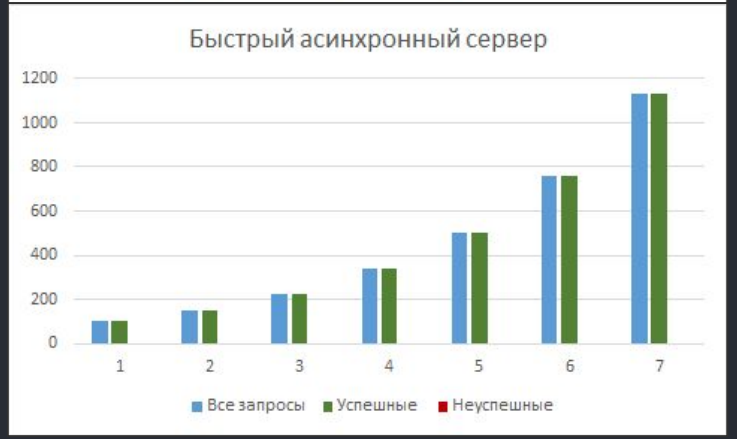
# Как работает честная асинхронность?

- 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)

# 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

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

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


private async Task WaitOneSecond()
{
    await Task.Delay(TimeSpan.FromSeconds(1));
}
```

# ThreadPool starvation (истощение пула потоков)

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

# ThreadPool starvation: lock

проверить

# ThreadPool starvation

DEMO

# ValueTask

# ValueTask<T>

- Легковесные таски, 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 **ооочень** важен
- Но накладывает кучу ограничений, о них далее

# ValueTask<T>: Пример

```
private string? _top250;

public async ValueTask<string> GetTop250()
{
    if (_top250 is not null)
    {
        return _top250;
    }

    _top250 = await File.ReadAllTextAsync("top250.txt");
    return _top250;
}
```

# 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>`

# 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 необязательно, т. к. в нем нет контексте синхронизации

# Помните этот код? (Deadlock)

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


private async Task WaitOneSecond()
{
    await Task.Delay(TimeSpan.FromSeconds(1));
}
```

# Помните этот код? (Deadlock)

```
private void Button_Click(object sender, RoutedEventArgs e)
{
   WaitOneSecond().GetAwaiter().GetResult(); //больше нет дедлока

   MessageBox.Show("Это сообщение покажется :)"); //тут эксепшн
}



private async Task WaitOneSecond()
{
    await Task.Delay(TimeSpan.FromSeconds(1)).ConfigureAwait(false);
}
```

# Асинхронное высвобождение ресурсов

- Реализуйте интерфейс IAsyncDisposable для получения возможности асинхронного высвобождения ресурсов
- В методе DispoiseAsync вы сможете использовать асинхронные вызовы

# IAsyncDisposable: Пример

```
public class MailKitEmailSender : IEmailSender, IDisposable, IAsyncDisposable
{
    private readonly SmtpClient _smtpClient;

    public async ValueTask DisposeAsync()
    {
        if (_smtpClient.IsConnected)
        {
            await _smtpClient.DisconnectAsync(true);
        }
        _smtpClient.Dispose();
    }
}
```

# Постфикс Async

- В стандартной библиотеке к асинхронным методам принято добавлять постфикс Async
- Но при разработке веб-приложения на ASP.NET Core у асинхронных методов обычно постфикс Async не указывают
- Обычно постфикс Async уместен в случае, когда доступен API аналогичного синхронного метода:
  - `GetProducts`
  - `GetProductsAsync`

# Отмена

## CancellationToken

# Отмена: CancellationToken

- Дает возможность отменить операцию
- В отмене участвуют две стороны: источник и получатель
- Источник (инициатор отмены) выпускает токен отмены через объект CancellationTokenSource
- Получатель проверяет, не вызвана ли отмена у маркера (CancellationToken)

# CancellationToken: Потребление

```
void DoHeavyJob(CancellationToken cancellationToken)
{
    for (int i = 0; i < 100_000_000; i++)
    {
        cancellationToken.ThrowIfCancellationRequested();
        NotifyUser(i);
    }
}
```

# CancellationToken: Выпуск и отмена

```
using var cts = new CancellationTokenSource(TimeSpan.FromSeconds(3));
try
{
    DoHeavyJob(cts.Token);
}
catch (OperationCanceledException)
{
    Console.WriteLine("Операция отменена");
}
```

Или так:

```
cts.Cancel();
```

# Отмена: CancellationToken

- Многие асинхронные API поддерживают CancellationToken, поэтому обеспечение отмены обычно сводится к простой передаче маркера
- Как правило, если ваш метод вызывает функции API, получающие CancellationToken, то ваш метод также должен получать CancellationToken и передавать его всем функциям API, которые его поддерживают

# Отмена: CancellationToken

- В ASP.NET Core есть возможность получения токена отмены запроса через параметр метода действия
- Это дает возможность вызывающему модулю отменить действие
  - Например, если браузер остановит запрос, то сработает отмена и в CancellationToken
- В некоторых компаниях такой подход является обязательным

# DEMO

CancellationToken в запросах ASP.NET Core

# Отмена в Parallel

Параллельные методы поддерживают эту возмож-

ность посредством получения экземпляра ParallelOptions. Установка

CancellationToken для экземпляра ParallelOptions выполняется так:

```
void RotateMatrices(IEnumerable<Matrix> matrices, float degrees,
CancellationToken token)
{
Parallel.ForEach(matrices,
new ParallelOptions { CancellationToken = token },
matrix => matrix.Rotate(degrees));
}
```

# Отмена в PLINQ

В Parallel LINQ (PLINQ) также предусмотрена встроенная поддержка отмены с оператором WithCancellation:

```
IEnumerable<int> MultiplyBy2(IEnumerable<int> values,
CancellationToken cancellationToken)
{
return values.AsParallel()
.WithCancellation(cancellationToken)
.Select(item => item * 2);
}
```

# IProgress<T>

- Используйте типы IProgress<T> и Progress<T>
- Ваш async-метод должен получать аргумент IProgress<T>
- T — тип прогресса, о котором вы хотите сообщать

```
async Task MyMethodAsync(IProgress<double> progress = null)
{
bool done = false;
double percentComplete = 0;
while (!done)
{
...
progress?.Report(percentComplete);
}
}
```

# IProgress<T>

Пример использования в вызывающем коде:

```
async Task CallMyMethodAsync()
{
var progress = new Progress<double>();
progress.ProgressChanged += (sender, args) =>
{
...
};
await MyMethodAsync(progress);
}
```

# 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

# Домашнее задание

1. Сделайте сервис отправки писем через EmailKit полностью асинхронным
2. Упражнение: Напишите асинхронный метод, который будет считывать содержимое файлов, имена которых переданы в params и возвращать ВСЕ строки