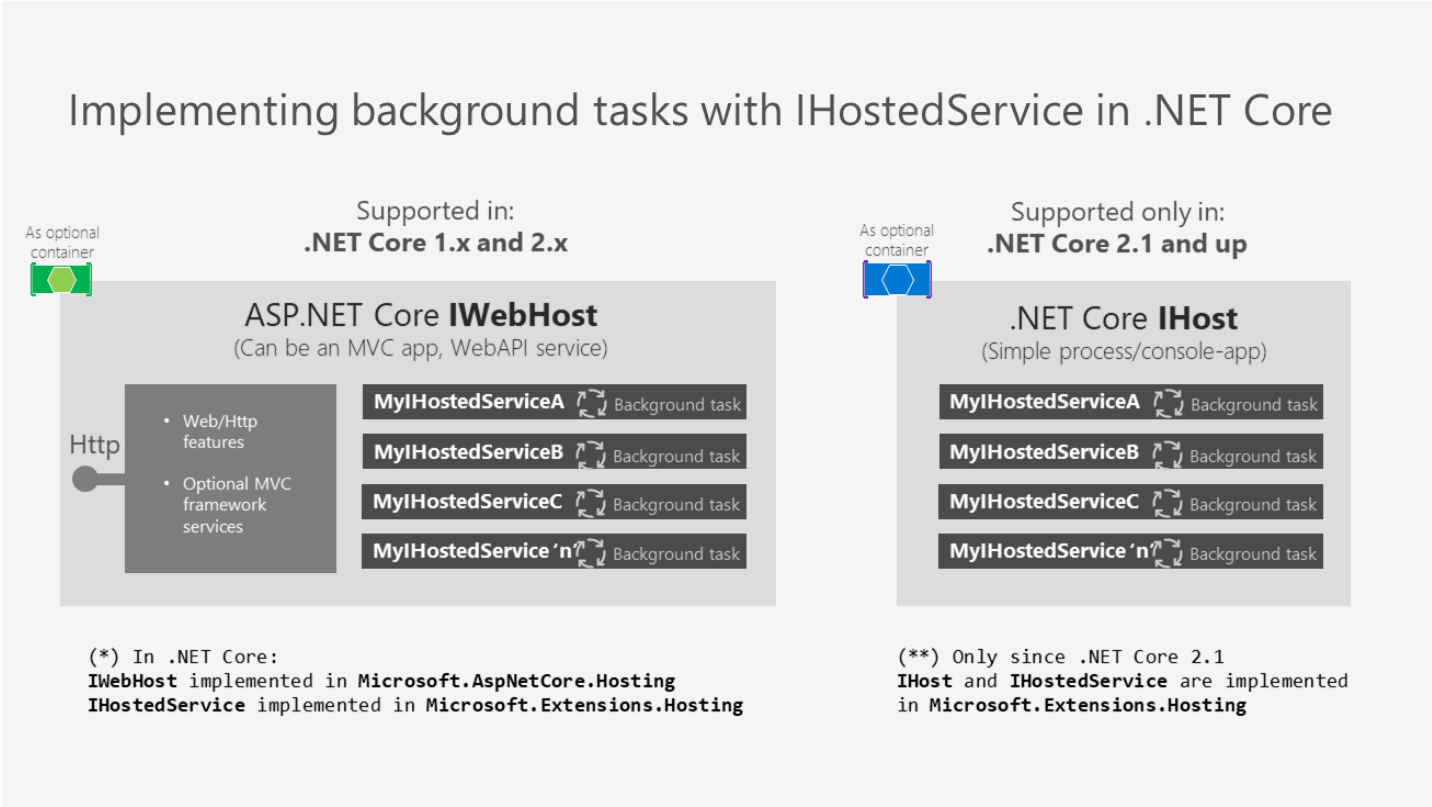
Запуск фоновых задач в asp.net core

10 мин 51K

Программирование*, .NET*, ASP*, C#*

Туториал

Небольшой обзор стандартных средств запуска бэкграунд-задач в аспнет приложениях — что есть, чем отличается, как пользоваться. Встроенный механизм запуска таких задач строится вокруг интерфейса IHostedService и метода-расширения для IServiceCollection — AddHostedService. Но есть несколько способов реализовать фоновые задачи через этот механизм (и ещё несколько неочевидных моментов поведения этого механизма).



Зачем запускать фоновые задачи

Есть 2 глобальных сценария фоновых задач:

- Однократный запуск фоновой задачи при старте приложения с ожиданием начала обработки запросов или до него. Например, можно проводить миграцию данных до начала обработки запросов, прогревать кэши или другие части приложения для решения проблемы обработки первых запросов. Ещё может понадобится дождаться старта приложения — чтобы зажечь beacon сервиса в service discovery, получив информацию о прослушиваемых адресах

- Периодический регулярный запуск фоновой задачи — это может быть health check, отправка телемитрии сервиса, инвалидация кэша

aspnet позволяет отделить фоновые сервисы и переиспользовать их в разных приложениях, что даст универсальный механизм для таких операций в разных сервисах. Во всех примерах ниже для регистрации нового фонового сервиса используется метод `ServiceCollectionHostedServiceExtensions.AddHostedService`:

```
services.AddHostedService<MyHostedService>();
```

Собственная реализация IHostedService

Интерфейс IHostedService предоставляет 2 метода:

```
public interface IHostedService
{
    // Вызывается, когда приложение готово запустить фоновую службу
    Task StartAsync(CancellationToken stoppingToken);
    // Вызывается, когда происходит нормальное завершение работы узла приложения.
    Task StopAsync(CancellationToken stoppingToken);
}
```

Что важно знать при реализации интерфейса? Все `IHostedService` запускаются последовательно, а вызов `StartAsync` блокирует запуск остальной части приложения. Поэтому в `StartAsync` не должно быть длинных блокирующих операций, если вы только действительно не хотите отложить запуск приложения до завершения этой операции (например, при миграции БД):

Именно это является особенностью и мотивацией реализовать фоновые операции через собственную реализацию `IHostedService` — если вам нужен полный контроль над запуском и остановкой фонового сервиса. Если это не так важно , то достаточно отнаследоваться от класса `BackgroundService`.

Ещё пара важных для реализации моментов:

- У `CancellationToken` в `StopAsync` есть по-умолчанию 5 секунд для корректного завершения

> Как правильно заметил в комментарии @kuda78— `CancellationToken` в StopAsync общий для всего хоста и сервисов, и HostedSerice останавливаются последовательно. То есть 5 секунд на завершение даётся всему хосту, а не каждому сервису. Если какой-то из `IHostedSerice` выполнял `StopAsync` 4 секунды, то следующим фоновым сервисам останется на корректное завершение только одна. Если приложению и фоновым сервисам точно необходимо больше времени для корректного завершения работы, то таймаут можно переопределить, задав явно значение HostOptions.ShutdownTimeout.

- StopAsync может вообще не быть вызван при неожиданном завершении приложения. Поэтому, например, недостаточно гасить beacon только в этом методе.

Общая реализация фоновой периодической задачи в этом случае может выглядеть примерно так:

```
public class MyHostedService : IHostedService
{
    private readonly ISomeBusinessLogicService someService;

    public MyHostedService(ISomeBusinessLogicService someService)
    {
        this.someService = someService;
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        // Не блокируем поток выполнения: StartAsync должен запустить выполнение фоновой зада
        DoSomeWorkEveryFiveSecondsAsync(cancellationToken);
        return Task.CompletedTask;
    }

    private async Task DoSomeWorkEveryFiveSecondsAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            try
            {
                await someService.DoSomeWorkAsync();
            }
            catch (Exception ex)
            {
                // обработка ошибки однократного неуспешного выполнения фоновой задачи
            }

            await Task.Delay(5000, stoppingToken);
        }
```

```
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        // Если нужно дождаться завершения очистки, но контролировать время, то стоит предусм
        await someService.DoSomeCleanupAsync(cancellationToken);
        return Task.CompletedTask;
    }
}
```

Наследование от BackgroundService

BackgroundService — это абстрактный класс, котоырй реализует `IHostedService`, сам обрабатывает запуск и остановку, предоставляя 1 абстрактный метод `ExecuteAsync`:

```
public abstract class BackgroundService : IHostedService, IDisposable
{
    private Task _executingTask;
    private readonly CancellationTokenSource _stoppingCts = new CancellationTokenSource();

    protected abstract Task ExecuteAsync(CancellationToken stoppingToken);

    public virtual Task StartAsync(CancellationToken cancellationToken)
    {
        _executingTask = ExecuteAsync(_stoppingCts.Token);
        return _executingTask.IsCompleted ? _executingTask : Task.CompletedTask;
    }

    public virtual async Task StopAsync(CancellationToken cancellationToken)
    {
        if (_executingTask == null)
            return;

        try
        {
            _stoppingCts.Cancel();
        }
        finally
        {
            await Task.WhenAny(_executingTask, Task.Delay(Timeout.Infinite, cancellationToken
        }
    }

    public virtual void Dispose() => _stoppingCts.Cancel();
}
```

`StartAsync` и `StopAsync` всё ещё можно перегрузить. Реализация фоновых задач через `BackgroundService` подходит для всех сценариев, где не нужно блокировать запуск приложения до завершения выполнения операции.

Общая реализация фоновой периодической задачи:

```
public class MyHostedService : BackgroundService
{
    private readonly ISomeBusinessLogicService someService;

    public MyHostedService(ISomeBusinessLogicService someService)
    {
        this.someService = someService;
    }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        // Выполняем задачу пока не будет запрошена остановка приложения
        while (!stoppingToken.IsCancellationRequested)
        {
            try
            {
                await someService.DoSomeWorkAsync();
            }
            catch (Exception ex)
            {
                // обработка ошибки однократного неуспешного выполнения фоновой задачи
            }

            await Task.Delay(5000);
        }

        // Если нужно дождаться завершения очистки, но контролировать время, то стоит предусм
        await someService.DoSomeCleanupAsync(cancellationToken);
    }
}
```

Когда и как запускается IHostedService

Занятный факт, правильный ответ — "зависит". От версии .net.

В .NET Core 2.x `IHostedService` запускались после конфигурирования и старта Kestrel, то есть после того, как приложение начинает слушать порты для приема запросов. Это значит, что, например, мы можем получить в фоновом сервисе объект `IServer` и `IServerAddressesFeature` и быть уверенными, что в момент запуска фонового сервиса список прослушиваемых адресов будет уже настроен. Ещё это значит, что на момент запуска

`IHostedService` приложение уже может отвечать на запросы клиентов, поэтому нельзя гарантировать, что на момент обработки запроса какой-то из `IHostedService` уже запущен.

В .NET Core 3.0 с переходом на новую абстракцию `IHost` (на самом деле универсальный узел появился уже в .net core 2.1) поведение изменилось — теперь Kestrel начал запускаться как отдельный `IHostedService` последним после всех остальных `IHostedService`. Фактически фоновые сервисы запускаются до метода `Statup.Configure()`. Теперь можно гарантировать, что на момент начала прослушивания портов и обработки запросов все другие фоновые сервисы запущены, а ещё можно не начинать обработку запросов до завершения запуска одного из фоновых сервисов с помощью переопределения `StartAsync`.

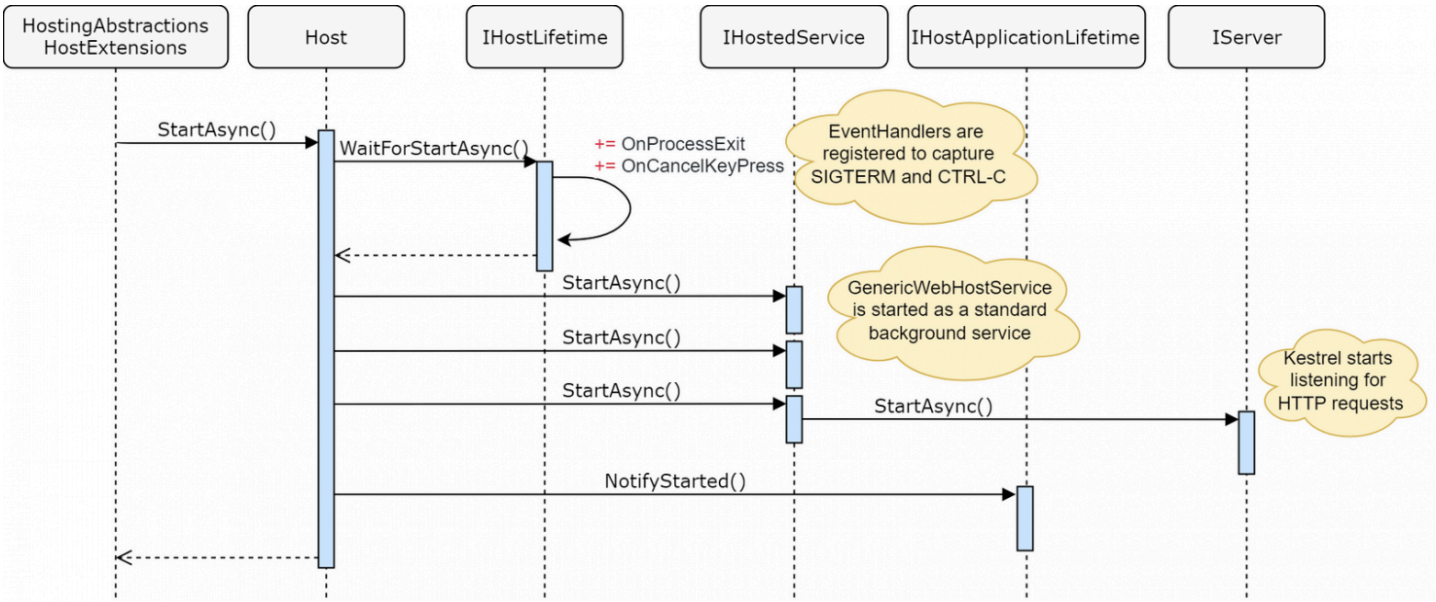


Иллюстрация от Andrew Lock https://twitter.com/andrewlocknet

В .NET 6 всё снова немного поменялось. Появился Minimal hosting API, в котором нет дополнительных абстрацией в виде Startup.cs, а приложение конфигурируется явно с помощью нового класса WebApplication. Тут надо отметить, что новое апи включено по-умолчанию в шаблон аспнет приложения, поэтому для новых проектов из коробки будет использоваться именно оно. Все `IHostedService` в этом случае запускаются, когда вы вызываете `WebApplication.Run()`, то есть, уже после того, как вы настроили приложение и список прослушиваемых адресов. Подробнее об этом написано в issue на github.

Фактически это значит, что поведение и доступные в `IHostedService` параметры могут меняться в зависимости от версии .net и способа хостинга, и не может полагаться внутри сервиса на то, что Kestrel уже сконфигурирован и запущен. Поэтому, если фоновый сервис работает с конфигурацией Kestrel, то нужен способ дождаться его запуска внутри `IHostedService`.

Ожидание запуска Kestrel внутри IHostedService

В asp.net core начиная с версии 3.0 появился сервис, который позволяет получить уведомления о том, что приложение завершило запуск и начало обрабатывать запросы — это IHostApplicationLifetime.

```
public interface IHostApplicationLifetime
{
    CancellationToken ApplicationStarted { get; }
    CancellationToken ApplicationStopping { get; }
    CancellationToken ApplicationStopped { get; }
    void StopApplication();
}
```

`CancellationToken` даёт удобный механизм безопасного запуска колбеков при возникновении события:

```
lifetime.ApplicationStarted.Register(() => DoSomeAction());
```

Благодаря этому мы можем дождаться запуска приложения. Но в ожидании запуска нам нужно обработать ситуацию, когда возникают проблемы с стартом — тогда приложение никогда не запустится, а ожидающий старта метод не завершится. Чтобы это исправить достататочно ждать не только `ApplicationStarted`, но и обрабаывать событие для `stoppingToken`, приходящего в `ExecuteAsync`. Вот как будет выглядеть полный пример фонового сервиса, который ожидает запуск приложения и корректно обрабатывает ошибки запуска:

```
public class MyHostedService : BackgroundService
{
    private readonly ISomeBusinessLogicService someService;
    private readonly IHostApplicationLifetime lifetime;

    public MyHostedService(ISomeBusinessLogicService someService, IHostApplicationLifetime li
    {
        this.lifetime = lifetime;
        this.someService = someService;
    }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
       if (!await WaitForAppStartup(lifetime, stoppingToken))
            return;

        // Приложение запущено и готово к обработке запросов

        // Выполняем задачу пока не будет запрошена остановка приложения
        while (!stoppingToken.IsCancellationRequested)
        {
            try
            {
                await someService.DoSomeWorkAsync();
            }
```

```
            catch (Exception ex)
            {
                // обработка ошибки однократного неуспешного выполнения фоновой задачи
            }

            await Task.Delay(5000);
        }

        // Если нужно дождаться завершения очистки, но контролировать время, то стоит предусм
        await someService.DoSomeCleanupAsync(stoppingToken);
    }

    static async Task<bool> WaitForAppStartup(IHostApplicationLifetime lifetime, Cancellation
    {
        // ? Создаём TaskCompletionSource для ApplicationStarted
        var startedSource = new TaskCompletionSource();
        using var reg1 = lifetime.ApplicationStarted.Register(() => startedSource.SetResult()

        // ? Создаём TaskCompletionSource для stoppingToken
        var cancelledSource = new TaskCompletionSource();
        using var reg2 = stoppingToken.Register(() => cancelledSource.SetResult());

        // Ожидаем любое из событий запуска или запроса на остановку
        Task completedTask = await Task.WhenAny(startedSource.Task, cancelledSource.Task).Con

        // Если завершилась задача ApplicationStarted, возвращаем true, иначе false
        return completedTask == startedSource.Task;
    }
}
```

Обработка исключений в IHostedService

Запуск `IHostedService` происходит в методе Host.StartAsync (код в .net 6):

```
foreach (IHostedService hostedService in _hostedServices)
{
    // Fire IHostedService.Start
    await hostedService.StartAsync(combinedCancellationToken).ConfigureAwait(false);

    if (hostedService is BackgroundService backgroundService)
    {
        _ = TryExecuteBackgroundServiceAsync(backgroundService);
    }
}
```

Если во время запуска фонового сервиса в методе `StartAsync` происходит исключение, то оно не обрабатывается снаружи и запуска хоста приложения прерывается. Если же в `IHostedService.StartAsync` запускается асинхронная операция без ожидания, то исключения из такого метода никогда не будут возвращены наружу и хост не узнают об ошибке в фоновом сервисе. В этом случае следует предусмотреть самостоятельную обработку исключений.

В наследниках `BackgroundService` поведение при необработанных исключениях в методе `ExecuteAsync` отличается в зависимости от версии .net.

В .NET до 6.0 необработанные исключения в `BackgroundService` просто терялись, а приложение продолжало работу. Начиная с .NET 6 при по-умолчанию необработанное исключение из `BackgroundService` будет залогировано, а приложение завершит работу. За это отвечает тот самый метод `TryExecuteBackgroundServiceAsync`, который выполняется для наследников `BackgroundService`:

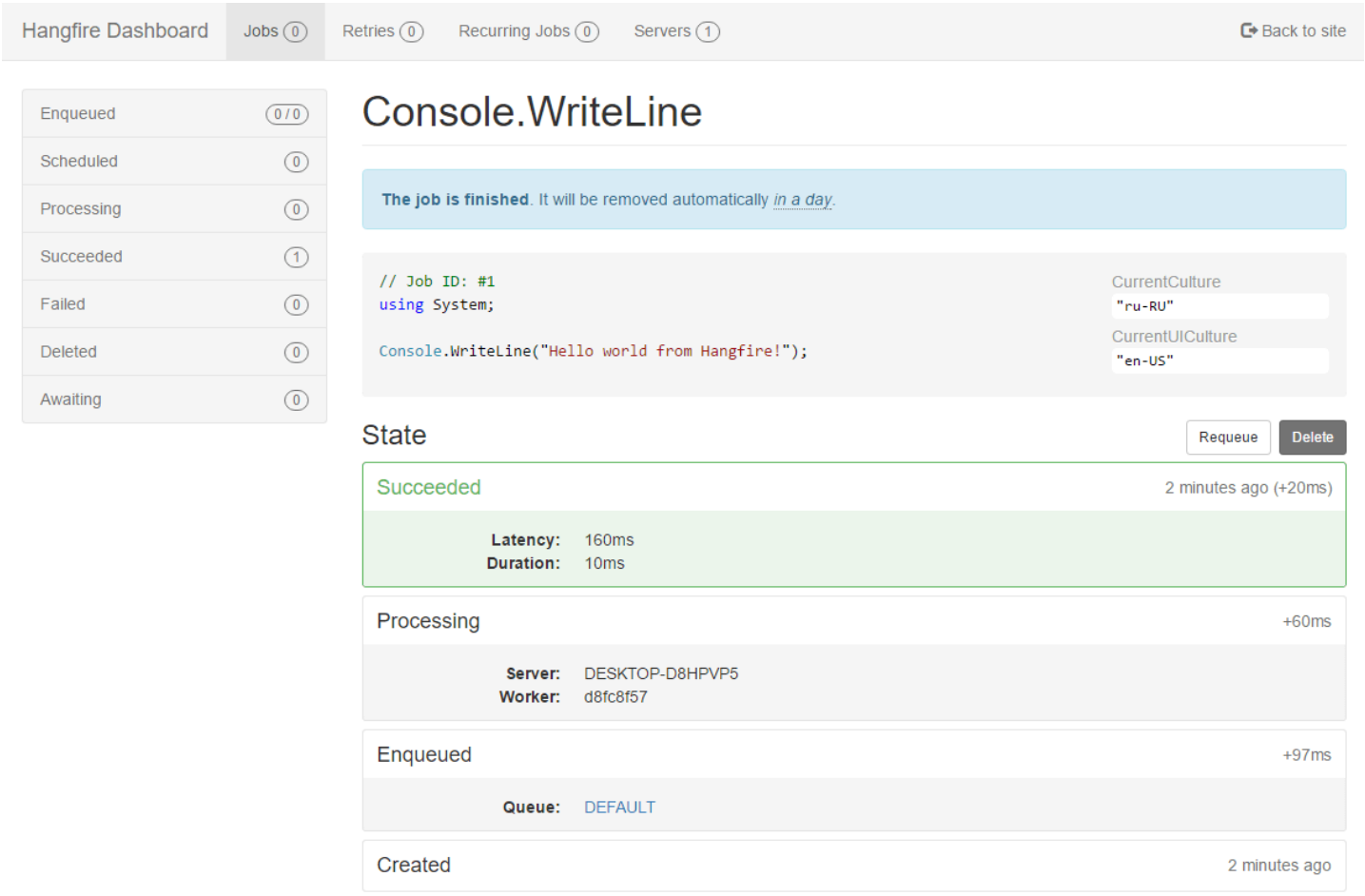
```
private async Task TryExecuteBackgroundServiceAsync(BackgroundService backgroundService)
{
    // backgroundService.ExecuteTask может остаться неинициализированным
    //(например, если в наследнике BackgroundService при переопределении StartAsync не был вы
    Task backgroundTask = backgroundService.ExecuteTask;
    if (backgroundTask == null)
        return;

    try
    {
        await backgroundTask.ConfigureAwait(false);
    }
    catch (Exception ex)
    {
        // Когда хост останавливается, он посылает сигнал для остановки фоновых служб.
        // Это не является ошибочной ситуацией, поэтому логировать это как ошибку не нужно.
        if (_stopCalled && backgroundTask.IsCanceled && ex is OperationCanceledException)
            return;

        _logger.BackgroundServiceFaulted(ex);
        if (_options.BackgroundServiceExceptionBehavior == BackgroundServiceExceptionBehavior
        {
            _logger.BackgroundServiceStoppingHost(ex);
            _applicationLifetime.StopApplication();
        }
    }
}
```

Чтобы переопределить поведение и не останавливать хост в случае исключений в `BackgroundService` можно при конфигурации сервисов использовать настройку HostOptions.BackgroundServiceExceptionBehavior:

```
Host.CreateBuilder(args)
    .ConfigureServices(services =>
    {
        services.Configure<HostOptions>(hostOptions =>
        {
            hostOptions.BackgroundServiceExceptionBehavior = BackgroundServiceExceptionBehavi
        });
    });
```

Сторонние библиотеки для фоновых задач

Ещё одно популярное (57 миллионов скачиваний) решение для фоновых задач в дотнет — это Hangfire, библиотека для настройки, запуска и хранения фоновых задач с бесплатной версией для коммерческого использования, большим количеством настроек и отдельной админкой задач.



Что почитать о фоновых задачах в asp.net

Материалы собраны из статей Microsoft docs, статей Andew Lock и выступления Scott Sauber на Rome .NET Conference:

- Microsoft docs: Implement background tasks in microservices with IHostedService and the BackgroundService class
- Jeow Li Huan: Background tasks with hosted services in ASP.NET Core
- Scott Sauber: Rome .NET Conference – The Background on Background Tasks in .NET 6
- Andrew Lock: Running async tasks on app startup in ASP.NET Core 3.0
- Andew Lock: Controlling IHostedService execution order in ASP.NET Core 3.x
- Andrew Lock: Waiting for your ASP.NET Core app to be ready from an IHostedService in .NET 6
- Breaking changes: Unhandled exceptions from a BackgroundService
- Andrew Lock: Extending the shutdown timeout setting to ensure graceful IHostedService shutdown

Теги: c#, asp.net core, background jobs, .net, .net 6.0

Хабы: Программирование, .NET, ASP, C#

88 Карма 0 Рейтинг

Вадим Мартынов @Vadimyan

Программист