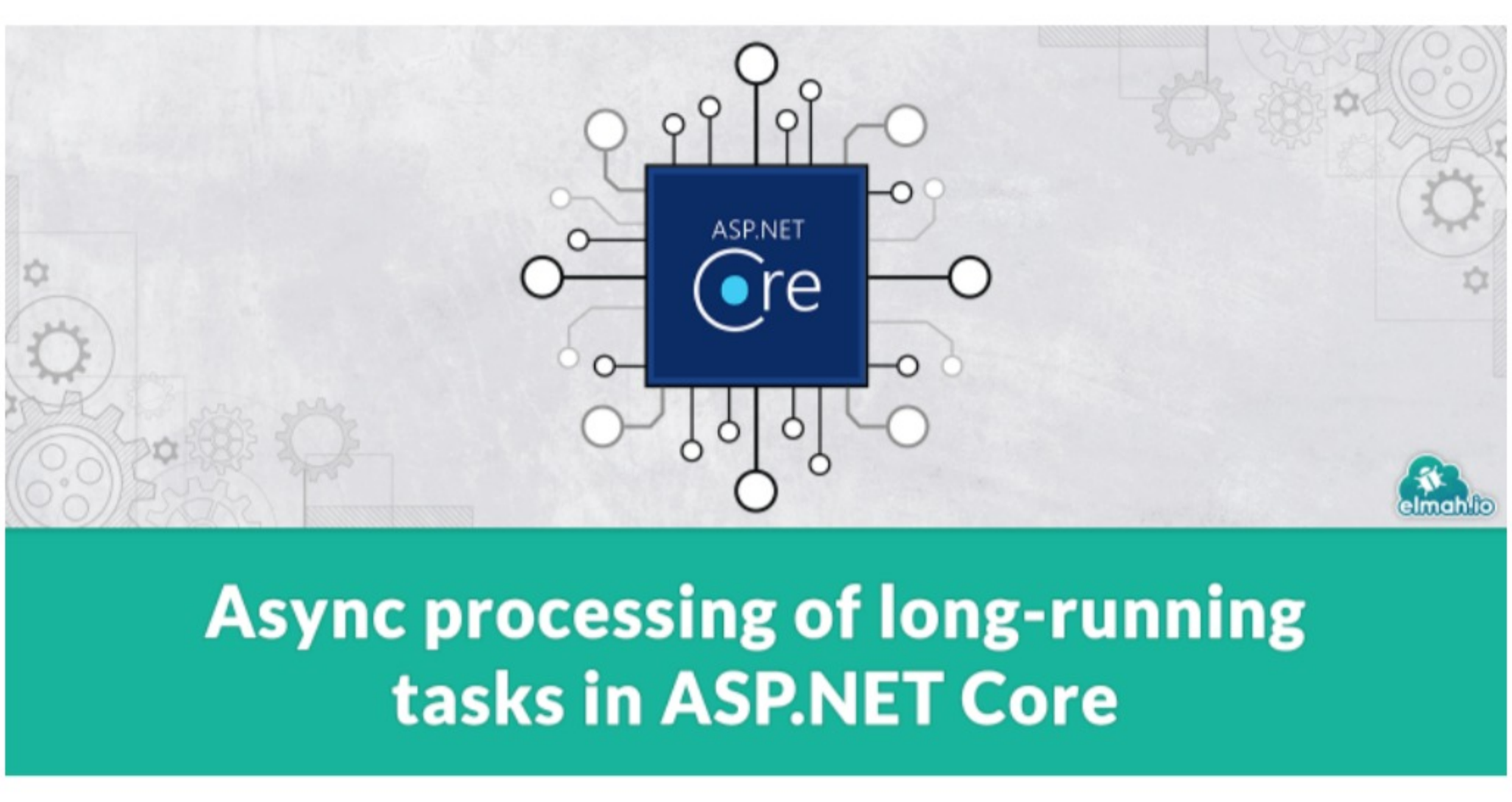
Async processing of long-running tasks in ASP.NET Core

Facebook Twitter LinkedIn Email

Written by Thomas Ardal, September 14, 2021

Sometimes, invoking an API endpoint needs to trigger a long-running task. Examples of this could be invoking an external and slow API or sending an email, which you don't want the caller of your API to wait for. There are multiple ways of implementing this using a message broker, a fire and forget API request, or something completely third. In this post, I'll show you how to implement async processing in ASP.NET Core, using a queue and the Background Worker feature.



> Before we begin let me make something clear. Background workers in ASP.NET Core are fine for handling minor workloads. There are some drawbacks, like jobs not being persisted on server restart. The solution proposed in this post can be a good v1 and when you feel ready for it, consider moving to a more robust approach like putting messages on a service bus or implementing a third-party library (check out Hangfire and Quartz.NET).

To understand how to process long-running tasks, let's start by creating an example and making it slooooow. Create a new ASP.NET Core application through Visual Studio, `dotnet`, or your preferred scaffolding engine. For this example, I have chosen the MVC template, but it could just as well be one of the other options.

Next, in the `HomeController.cs` file, create a new method to simulate a call to a slow running task:

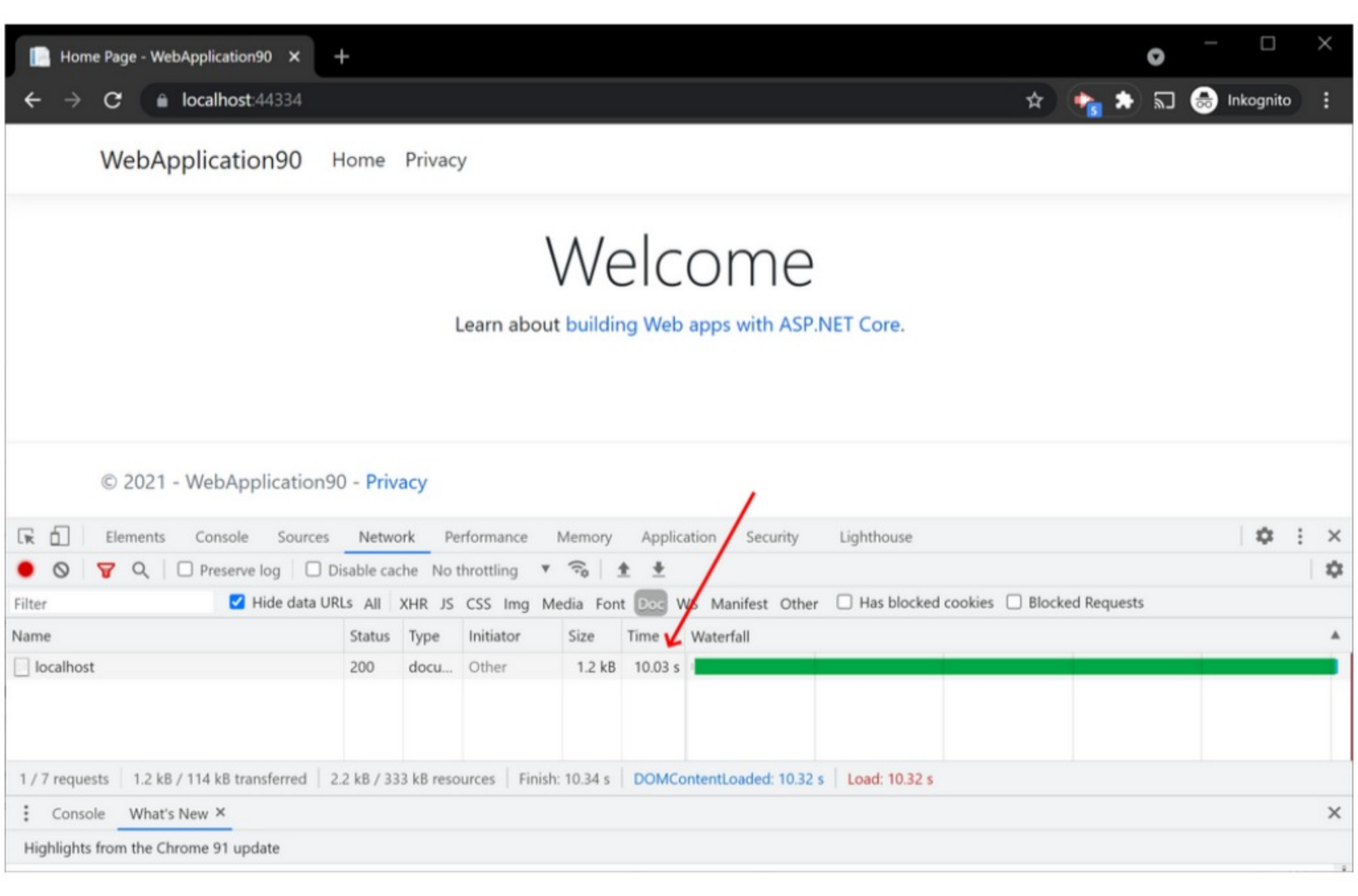
```
private async Task CallSlowApi()
{
    _logger.LogInformation($"Starting at {DateTime.UtcNow.TimeOfDay}");
    await Task.Delay(10000);
    _logger.LogInformation($"Done at {DateTime.UtcNow.TimeOfDay}");
}
```

For the demo, I'm waiting 10 seconds to simulate some work. The `Task.Delay` line would be replaced with some integration code in a real-life example. I have wrapped the code in information messages, which I can later inspect in Visual Studio or through the configured logger (maybe elmah.io?).

Then, invoke the `CallSlowApi` method from the `Index` method:

```
public async Task<IActionResult> Index()
{
    await CallSlowApi();
    return View();
}
```

Let's run the application and inspect the performance in Developer Tools:



As expected, loading the frontpage takes just above 10 seconds (10 seconds for the `Task.Delay` and 30 milliseconds to load the page).

Refactoring time! To process the message asynchronously, we'll implement a background worker in ASP.NET Core with an async queue in front. Let's start with the queue. Add a new class named `BackgroundWorkerQueue` and implementation like shown here:

```
public class BackgroundWorkerQueue
{
    private ConcurrentQueue<Func<CancellationToken, Task>> _workItems = new ConcurrentQueue<Func<Cance
    private SemaphoreSlim _signal = new SemaphoreSlim(0);

    public async Task<Func<CancellationToken, Task>> DequeueAsync(CancellationToken cancellationToken)
    {
        await _signal.WaitAsync(cancellationToken);
        _workItems.TryDequeue(out var workItem);

        return workItem;
    }

    public void QueueBackgroundWorkItem(Func<CancellationToken, Task> workItem)
    {
        if (workItem == null)
        {
            throw new ArgumentNullException(nameof(workItem));
        }

        _workItems.Enqueue(workItem);
        _signal.Release();
    }
}
```

It's a pretty simple implementation of a C#-based queue, using the `ConcurrentQueue` class from the `System.Collections.Concurrent` namespace. The `QueueBackgroundWorkItem` method will put a `Func` in the queue for later processing and the `DequeueAsync` method will pull a `Func` from the queue and return it.

Next, we need someone to execute the `Func` work items put on the queue. Add a new class named `LongRunningService` with the following implementation:

```
public class LongRunningService : BackgroundService
{
    private readonly BackgroundWorkerQueue queue;

    public LongRunningService(BackgroundWorkerQueue queue)
    {
        this.queue = queue;
    }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            var workItem = await queue.DequeueAsync(stoppingToken);

            await workItem(stoppingToken);
        }
    }
}
```

This is an implementation of an ASP.NET Core background service, which is indicated by extending `BackgroundService`. The service accepts the queue that we implemented in the last step and automatically dequeue and execute work items.

Both the `BackgroundWorkerQueue` and `LongRunningService` classes need to be registered with ASP.NET Core. Include the following code in the `ConfigureServices` method in the `Startup.cs` file:

```
services.AddHostedService<LongRunningService>();
services.AddSingleton<BackgroundWorkerQueue>();
```

That's it. All we need now is to refactor the `CallSlowApi` method. The `HomeController` need an instance of the `BackgroundWorkerQueue` class injected in its constructor:

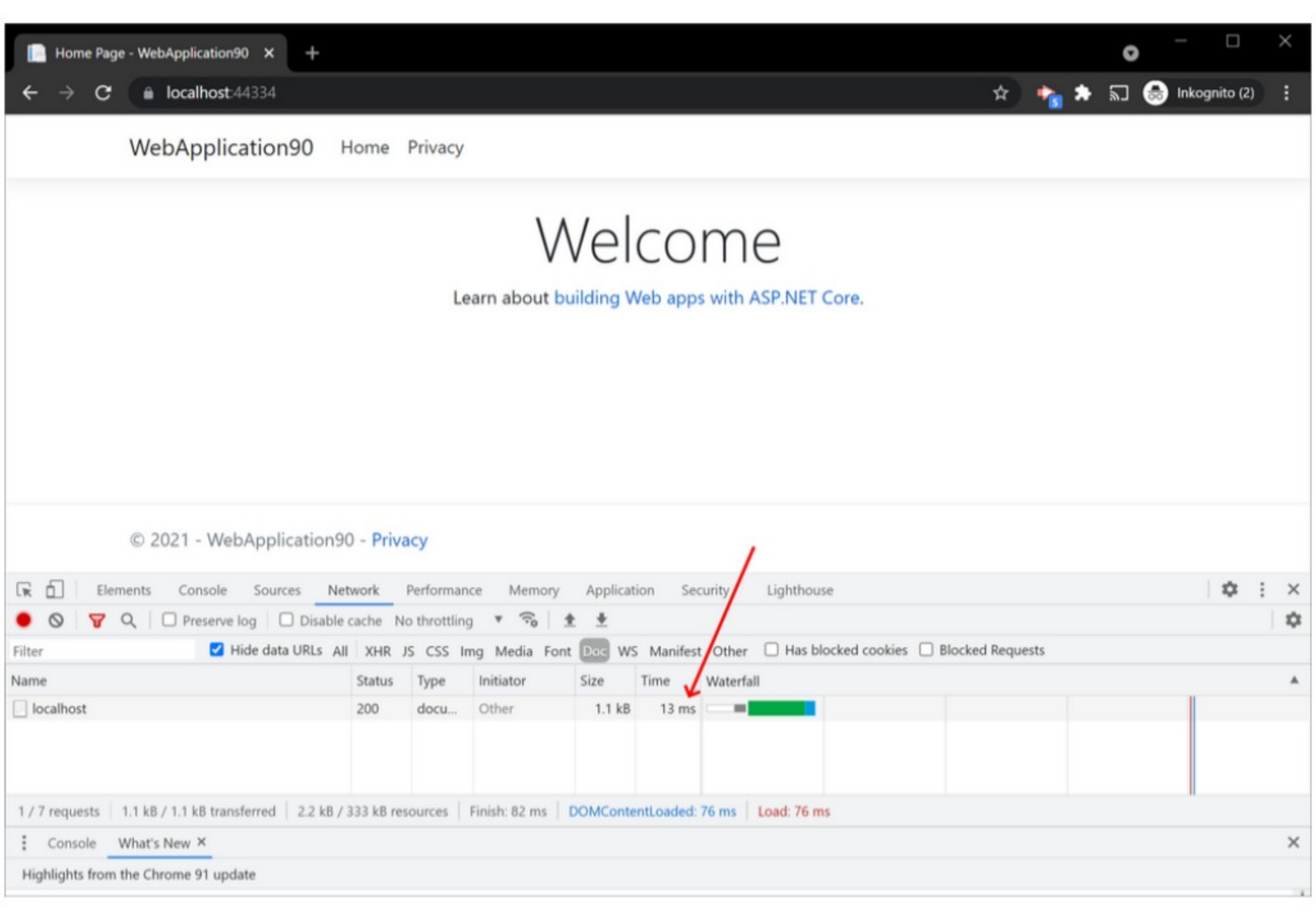
```
private readonly ILogger<HomeController> _logger;
private readonly BackgroundWorkerQueue _backgroundWorkerQueue;

public HomeController(ILogger<HomeController> logger, BackgroundWorkerQueue backgroundWorkerQueue)
{
    _logger = logger;
    _backgroundWorkerQueue = backgroundWorkerQueue;
}
```

The `Task.Delay` call can be moved inside a `Func` and handed off to the queue like this:

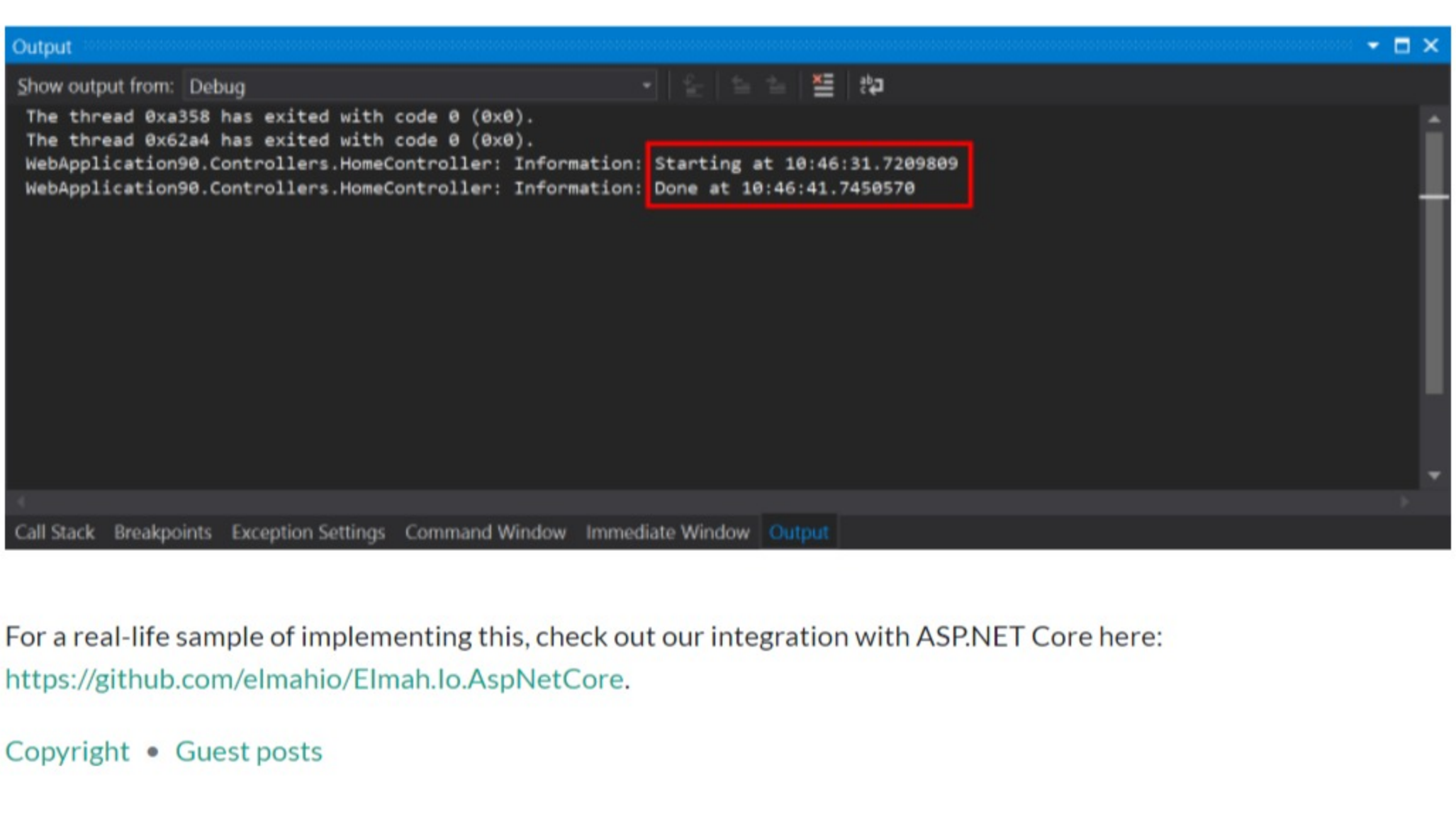
```
private async Task CallSlowApi()
{
    _logger.LogInformation($"Starting at {DateTime.UtcNow.TimeOfDay}");
    _backgroundWorkerQueue.QueueBackgroundWorkItem(async token =>
    {
        await Task.Delay(10000);
        _logger.LogInformation($"Done at {DateTime.UtcNow.TimeOfDay}");
    });
}
```

I simply moved the last two lines of the existing method inside the `Func` provided for the `QueueBackgroundWorkItem` method.

Launching the website is now snappy as ever:



To prove that the long-running task is still actually running, you can put a breakpoint after the call to `Task.Delay` or you can simply inspect the log output in Visual Studio:

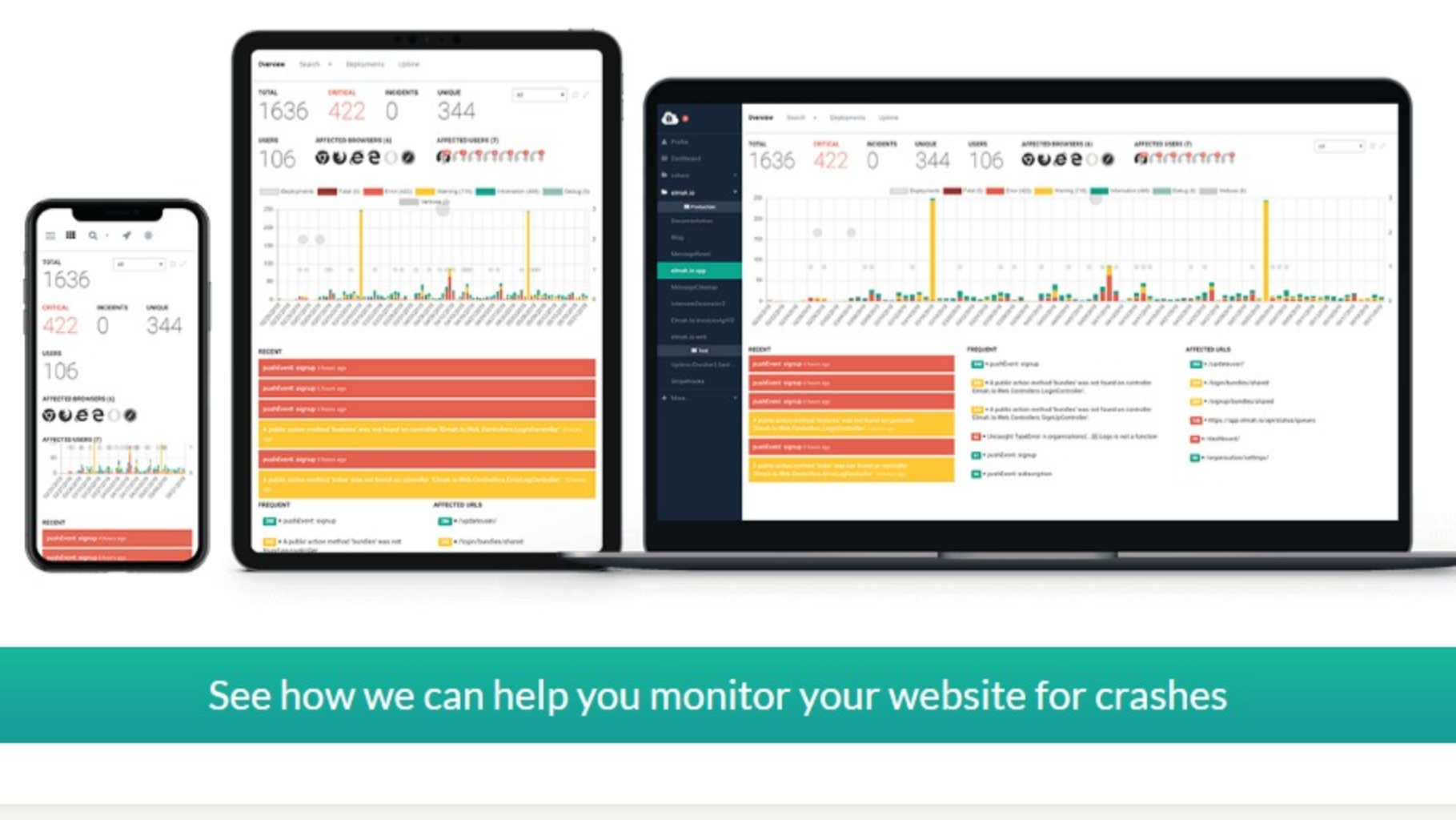


For a real-life sample of implementing this, check out our integration with ASP.NET Core here: https://github.com/elmahio/Elmah.Io.AspNetCore.

Copyright • Guest posts

← PREVIOUS POST
Debugging System.FormatException when launching ASP.NET Core

NEXT POST →
Ahead-Of-Time Compilation for Blazor Wasm



Search ...

Featured Posts

Download, parse, and store SSL certificates in C#
AppSettings in ASP.NET Core
Tuning performance on Azure websites using remote profiling
Export data from Google Analytics with .NET
The Ultimate ELMAH Tutorial
Logging in Unity

Blog Posts

APRIL 2022
ASP.NET Core 7 has built-in dark mode for error pages

MARCH 2022
Search videos through the YouTube Data API from C#
How to integrate with the Trello API from .NET/C#

FEBRUARY 2022
Collect and track Lighthouse results with elmah.io
Download, parse, and store SSL certificates in C#
Introducing the Dark Screen of Death

Show all posts