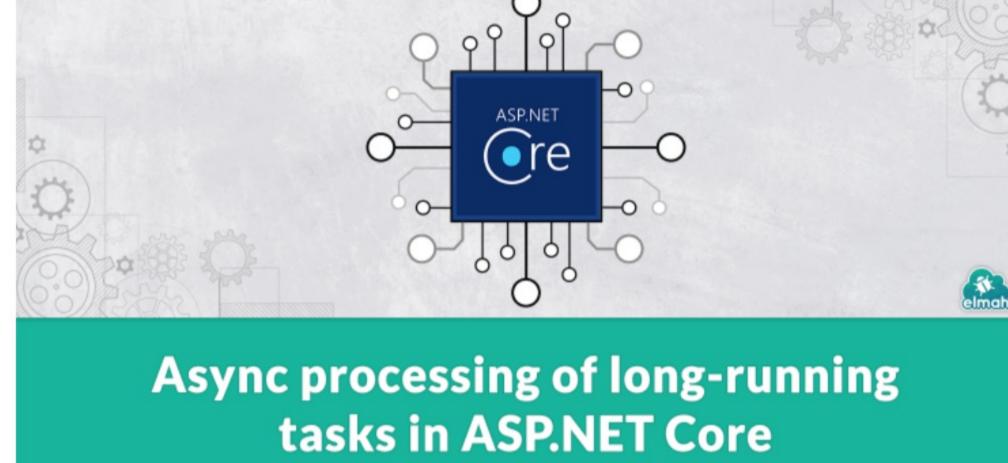
When your website feels sick, we've got your back!

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

f Facebook **▼** Twitter in LinkedIn 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 sloooooow. 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?).
```

public async Task<IActionResult> Index()

return View(); Let's run the application and inspect the performance in Developer Tools:

WebApplication90 Home Privacy

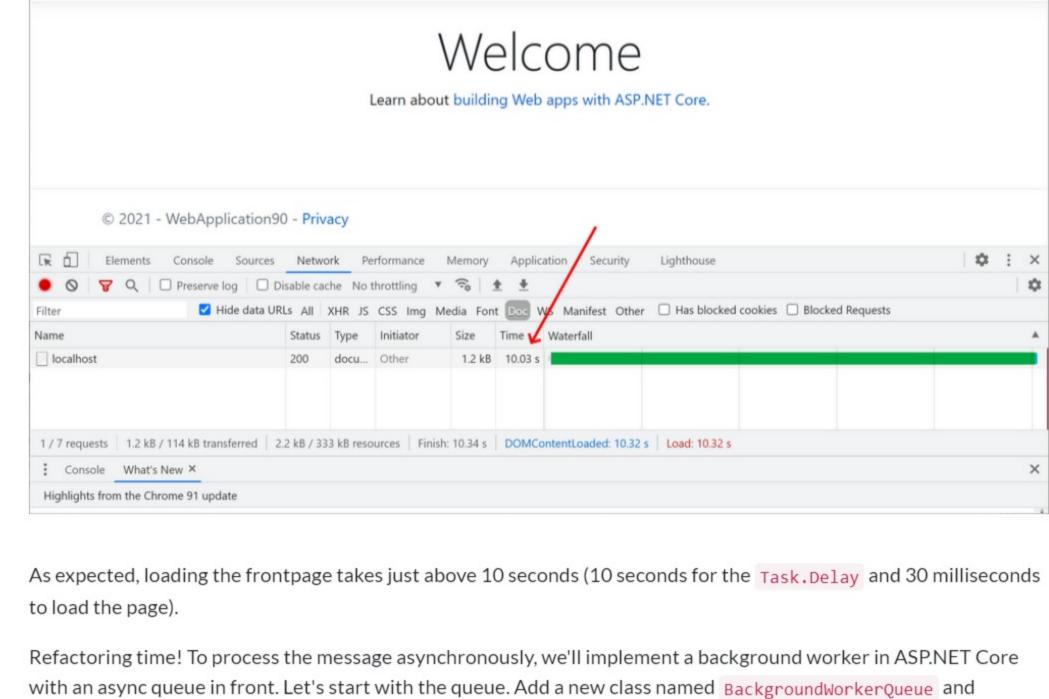
Then, invoke the CallSlowApi method from the Index method:

await CallSlowApi();

Home Page - WebApplication90 X +

C a localhost:44334

implementation like shown here:



public class BackgroundWorkerQueue

private ConcurrentQueue<Func<CancellationToken, Task>> _workItems = new ConcurrentQueue<Func<Cancel</pre> 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 LongRunningService(BackgroundWorkerQueue queue) this.queue = queue;

protected override async Task ExecuteAsync(CancellationToken stoppingToken)

public class LongRunningService : BackgroundService

private readonly BackgroundWorkerQueue queue;

BackgroundWorkerQueue class injected in its constructor:

backgroundWorkerQueue = backgroundWorkerQueue;

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

_logger = logger;

Launching the website is now snappy as ever:

© 2021 - WebApplication90 - Privacy

Home Page - WebApplication90 X

← → C 👜 localhost:44334

method.

```
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>();
```

private readonly ILogger<HomeController> _logger; private readonly BackgroundWorkerQueue _backgroundWorkerQueue; public HomeController(ILogger<HomeController> logger, BackgroundWorkerQueue backgroundWorkerQueue)

That's it. All we need now is to refactor the CallslowApi method. The HomeController need an instance of the

```
13
  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
```

Learn about building Web apps with ASP.NET Core.

🖈 🍖 🔝 🔝 Inkognito (2)

WebApplication90 Home Privacy Welcome

```
Elements Console Sources Network Performance Memory Application Security
        ▼ Q Preserve log Disable cache No throttling
                     ☑ Hide data URLs All XHR JS CSS Img Media Font Doc WS Manifest Other ☐ Has blocked cookies ☐ Blocked Requests
 localhost
 1 / 7 requests 1.1 kB / 1.1 kB transferred 2.2 kB / 333 kB resources Finish: 82 ms DOMContentLoaded: 76 ms Load: 76 ms
 Console What's New X
 Highlights from the Chrome 91 update
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:
                                                                                                                                   ▼ □ ×
                                                             Show output from: Debug
 The thread 0xa358 has exited with code 0 (0x0).
 The thread 0x62a4 has exited with code 0 (0x0).
 WebApplication90.Controllers.HomeController: Information: Starting at 10:46:31.7209809
 WebApplication90.Controllers.HomeController: Information: Done at 10:46:41.7450570
 Call Stack Breakpoints Exception Settings Command Window Immediate Window Output
```

Copyright • Guest posts

elmah.io: Error logging and Uptime Monitoring for your web apps

For a real-life sample of implementing this, check out our integration with ASP.NET Core here:

This blog post is brought to you by elmah.io. elmah.io is error logging, uptime monitoring, deployment tracking, and service heartbeats for your .NET and JavaScript applications. Stop relying on your users to notify you when something is wrong or dig through hundreds of megabytes of log files spread across servers. With elmah.io, we store all of your log messages, notify you through popular channels like email, Slack, and Microsoft Teams, and help you fix

← PREVIOUS POST

ASP.NET Core

errors fast.

https://github.com/elmahio/Elmah.lo.AspNetCore.

Debugging System.FormatException when launching

ovee o ganana

0065.0

Learn

API

See how we can help you monitor your website for crashes

Tour Blog App Store Docs

Terms of Use | Privacy Policy | Refund Policy

Company

About

Contact

elmah.io

Product

Features

Pricing

.NET DevOps **Customer Stories** For CEOs and CTOs .NET Microservices .NET Cloud Logging Alternatives ASP.NET Core Error Logging

Guides

in

NEXT POST →

Ahead-Of-Time Compilation for Blazor Wasm

Search ...

certificates in C#

Featured Posts

AppSettings in ASP.NET Core Tuning performance on Azure websites using remote profiling Export data from Google Analytics with .NET

Download, parse, and store SSL

The Ultimate ELMAH Tutorial Logging in Unity

newsletter and receive

Sign up for our

Join our newsletter

a free copy of our book .NET Web Application Logging Essentials Submit Enter your email

Follow elmah.io

Blog Posts

in

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#

Collect and track Lighthouse results with elmah.io

certificates in C#

Introducing the Dark Screen of Death

Show all posts

☆ 🔖 🖈 🖫 😸 Inkognito

FEBRUARY 2022 Download, parse, and store SSL

Sign Up Now no credit card required

Web.config Validator **Stack Trace Formatter** Web.config Transform Tester More Free Tools

Online Tools