Building Worker Services and Scheduling Runs in .NET Core

Siva V Jan 17, 2024 31.5k 0 2

Introduction

Worker services in .NET Core have emerged as a powerful mechanism for building background services that run continuously and perform various tasks. This blog provides a step-by-step guide on creating worker services, delving into their evolution, and elucidating the code snippets along with the execution process.

Evolution of Worker Services

Worker services represent a continuation of the .NET Core journey, evolving from the need to run background tasks efficiently. They inherit concepts from hosted services and console applications, offering a robust solution for scenarios where continuous execution is required, such as data processing, message queue consumption, or scheduled tasks.

Step 1. Create a Worker Service

Begin by creating a new worker service project using the .NET CLI

```
dotnet new worker -n MyWorkerService
```

This command generates the basic structure for a worker's service.

Step 2. Define Worker Logic

Navigate to the generated `Worker.cs` file. Here, you can define the logic that your worker service will execute continuously. For example.

```
public class Worker : BackgroundService
{
```

```
    private readonly ILogger<Worker> _logger;

    public Worker(ILogger<Worker> logger)
    {
        _logger = logger;
    }

    protected override async Task ExecuteAsync(CancellationToken stop
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.LogInformation("Worker running at: {time}", DateT
            await Task.Delay(1000, stoppingToken);
        }
    }
}
```

In this example, the worker logs a message every second. You can replace this logic with your specific requirements.

Step 3. Configure and Run the Worker

Navigate to the `Program.cs` file and configure the worker service within the `CreateHostBuilder` method

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureServices((hostContext, services) =>
        {
            services.AddHostedService<Worker>();
        });
```

This configuration registers the worker service to be managed by the host.

Step 4. Run the Worker Service

Build and run the worker service using the following commands

```
dotnet build
dotnet run
```

The worker service will start running, and you'll see log messages being generated at the specified interval.

Step 5. Execution Process

The `ExecuteAsync` method within the worker service is the entry point for the continuous execution process. It runs until the cancellation token signals a shutdown request. The `Task.Delay` ensures the specified delay between iterations.

Schedule Runs

Understanding how to schedule tasks at specific intervals or times enhances the versatility of worker services. Follow this step-by-step guide to incorporate scheduling into your .NET Core worker service.

Install Required Packages

To enable scheduling, you'll need the `Quartz` library. Install it using the following command

```
dotnet add package Quartz
```

Define a Scheduled Job

Create a class for the job you want to schedule. For example, a job that runs every hour

```
    public class HourlyJob : IJob
    {
        private readonly ILogger<HourlyJob> _logger;

        public HourlyJob(ILogger<HourlyJob> logger)
        {
            _logger = logger;
        }

        public async Task Execute(IJobExecutionContext context)
```

```
    {
        _logger.LogInformation("Hourly job executed at: {time}", Date
    }
}
```

Configure Quartz

In the `ConfigureServices` method of `Startup.cs`, configure Quartz to use the `JobScheduler`.

```
services.AddSingleton(provider =>
{
    var schedulerFactory = new StdSchedulerFactory();
    var scheduler = schedulerFactory.GetScheduler().Result;

    scheduler.Start();

    var job = JobBuilder.Create<HourlyJob>()
                        .Build();

    var trigger = TriggerBuilder.Create()
                                .WithIdentity("hourlyTrigger", "defaul
                                .StartNow()
                                .WithSimpleSchedule(x => x.WithInterva
                                .Build();

    scheduler.ScheduleJob(job, trigger);

    return scheduler;
});
```

This configuration sets up a job (`HourlyJob`) to run every hour.

Modify Worker Service

Update the `Worker.cs` to inject the Quartz scheduler and handle its lifecycle.

```
public class Worker : BackgroundService
{
    private readonly ILogger<Worker> _logger;
    private readonly IScheduler _scheduler;

    public Worker(ILogger<Worker> logger, IScheduler scheduler)
    {
        _logger = logger;
        _scheduler = scheduler;
    }

    protected override async Task ExecuteAsync(CancellationToken stop
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.LogInformation("Worker running at: {time}", DateT
            await Task.Delay(1000, stoppingToken);
        }
    }

    public override async Task StopAsync(CancellationToken cancellati
    {
        await _scheduler.Shutdown();
        await base.StopAsync(cancellationToken);
    }
}
```

Run the Application

Build and run the application using the following commands.

```
dotnet build
dotnet run
```

Now, you'll see the worker service logging every second and the hourly job executing at the specified interval.

Conclusion

Creating a worker service in .NET Core is a straightforward process, offering a versatile solution for background task execution. The evolution from hosted services and console applications has resulted in a framework that seamlessly integrates with the broader .NET ecosystem. By following this step-by-step guide, you can build and run your worker service, gaining hands-on experience in the world of background task management in .NET Core.

By integrating scheduling into your .NET Core worker service using Quartz, you can extend its functionality to include frequent runs, such as hourly tasks. This step-by-step guide demonstrates the seamless integration of Quartz and worker services, providing you with a powerful toolset for managing background tasks. As you explore this enhanced capability, consider adapting the scheduling logic to suit your specific requirements, such as daily, weekly, or custom intervals.

Happy coding!

worker service .net core

> Next Recommended Reading
>
> Create APIs Using .Net Core 3.1 | REST API in .Net Core 3.1 with Entity Framework

Siva V TOP 500

429 490.4k 1

With over 8 years of experience as a .NET Full Stack Lead Developer, I specialize in building scalable, efficient solutions using .NET Core and modern front-end technologies. I'm passionate about clean code, best practi... Read more

https://dotnet-fullstack-dev.blogspot.com/

0

Type your comment here and press Enter Key (Minimum 10 characters)