Il set up

La prima cosa che devi sapere è che hai bisogno di .NET Core 3.0 installato. Al momento della scrittura, .NET Core 3.1 è appena stato distribuito e Visual Studio dovrebbe comunque chiederti di eseguire l'aggiornamento. Ma se stai provando a farlo in un progetto .NET Core 2.X, non funzionerà.

Se ti piace creare progetti dalla riga di comando, devi creare un nuovo progetto di tipo "lavoratore":

dotnet new worker

Se sei una persona di Visual Studio come me, in realtà c'è un modello all'interno di Visual Studio che fa esattamente la stessa cosa.



In questo modo viene creato un progetto con essenzialmente due file. Avrai il tuo program.cs che è fondamentalmente il "bootstrapper" per la tua app. E poi hai qualcosa chiamato worker.cs che è dove va la logica del tuo servizio.

Dovrebbe essere abbastanza facile da individuare, ma per aggiungere servizi in background extra a questo programma da eseguire in parallelo, è sufficiente creare una nuova classe che eredita da BackgroundService:

public class MyNewBackgroundWorker : BackgroundService { protected override Task ExecuteAsync ( CancellationToken stoppingToken ) { // Fai qualcosa. } }

Quindi, nel nostro file program.cs, aggiungiamo semplicemente il worker alla nostra raccolta di servizi:

. ConfigureServices (( hostContext , services ) => {

services . AddHostedService < Worker > ();

services . AddHostedService < MyNewBackgroundWorker > (); });

AddHostedService è stato effettivamente nel framework per un bel po 'di tempo come un task runner di tipo "servizio in background" che in genere viene eseguito sotto la tua applicazione web. In realtà abbiamo già fatto un articolo sui [servizi ospitati in ASP.NET Core](https://dotnetcoretutorials.com/2019/01/13/hosted-services-in-asp-net-core/) , ma in questo caso il servizio ospitato è fondamentalmente l'intera app piuttosto che essere qualcosa che viene eseguito dietro le quinte della tua app web.

Esecuzione / debug della nostra applicazione

Fuori dagli schemi, il modello di lavoro ha un servizio in background che pompa semplicemente il datetime nella finestra della console. Premiamo semplicemente F5 su un'app nuova di zecca e vediamo cosa otteniamo.

info: CoreWorkerService.Worker [0]

Worker in esecuzione alle: 12/07/2019 08:20:30 +13: 00

info: Microsoft.Hosting.Lifetime [0]

Applicazione avviata. Premi Ctrl + C per spegnere.

Siamo operativi immediatamente! Possiamo lasciare la finestra della nostra console aperta per eseguire il debug dell'applicazione o chiudere la finestra per uscire. Rispetto all'inferno che abbiamo provato a eseguire il debug del nostro servizio Windows durante la creazione in modo "Microsoft", questo è come il paradiso.

Un'altra cosa da notare è che quello che abbiamo davvero davanti a noi è una piattaforma per scrivere applicazioni per console. Alla fine stiamo solo scrivendo il tempo nella finestra della console, ma lo stiamo anche facendo tramite Dependency Injection creando un worker ospitato. Possiamo usare questo contenitore DI anche per iniettare in repository, impostare ambienti, leggere la configurazione ecc.

L'unica cosa che non è ancora è un servizio Windows ...

Trasformare la nostra app in un servizio Windows

Dobbiamo aggiungere il seguente pacchetto alla nostra app:

Install-Package Microsoft.Extensions.Hosting.WindowsServices

Successivamente, vai al nostro file program.cs e modificalo aggiungendo una chiamata a "UseWindowsService ()".

public static IHostBuilder CreateHostBuilder ( string [] args ) => Host . CreateDefaultBuilder ( args ) . ConfigureServices (( hostContext , services ) => {

services . AddHostedService < Worker > (); }). UseWindowsService ();

E questo è tutto!

L'esecuzione della nostra applicazione normalmente è la stessa e tutto funziona come prima. La grande differenza è che ora possiamo installare tutto come servizio.

Per fare ciò, prima dobbiamo pubblicare la nostra applicazione. Nella directory del progetto eseguiamo:

dotnet publish -r win-x64 -c Release

Nota nel mio caso, sto pubblicando per Windows X64 che generalmente avverrà durante la distribuzione di un servizio **Windows** .

Quindi tutto ciò che dobbiamo fare è eseguire il programma di installazione del servizio Windows standard. Non è specifico di .NET Core ma fa invece parte di Windows:

sc create TestService BinPath = C : \ full \ path \ to \ publish \ dir \ WindowsServiceExample . EXE

Come sempre, gli altri comandi a tua disposizione (incluso l'avvio del servizio) sono:

sc avvia TestService

sc stop TestService

sc elimina TestService

E controllando il nostro pannello dei servizi:

https://dotnetcoretutorials.com/wp-content/uploads/2019/12/TestServiceInsalled.png

Ha funzionato!

Installazione su Linux

Ad essere onesti, non ho molta esperienza con Linux. Ma l'essenza generale è ...

Invece di installare Microsoft.Extensions.Hosting.WindowsServices  , è necessario installare Microsoft.Extensions.Hosting.Systemd . E poi invece di chiamare UseWindowsService ()   chiamerai invece UseSystemd () .

Ovviamente i tuoi comandi di pubblicazione e installazione di dotnet varieranno, ma più o meno puoi creare un "servizio Windows" che funzionerà anche su Linux!

Microsoft vs Topshelf vs .NET Core Workers

Quindi ora abbiamo esaminato 3 diversi modi per creare servizi Windows. Probabilmente sei seduto lì a dire "Beh ... Quale dovrei scegliere?". Immediatamente, eliminiamo il primo modo di procedere della vecchia scuola di Microsoft. È infernale eseguire il debug e non ha davvero nulla da fare.

Questo ci lascia con Topshelf e .NET Core worker. A mio parere, mi piace .NET Core Workers per adattarsi senza sforzo all'ecosistema .NET Core. Se stai già sviluppando in ASP.NET Core, allora tutto ha senso creare un lavoratore. Inoltre, quando crei un BackgroundService, puoi effettivamente sollevarlo e spostarlo per eseguirlo all'interno di un sito Web ASP.NET Core in qualsiasi momento, il che è molto utile. L'unico inconveniente è l'installazione. Dover usare i comandi SC può essere incredibilmente frustrante a volte e Topshelf ha sicuramente battuto lì.

Topshelf in generale è molto facile da usare e ha il miglior processo di installazione per i servizi di Windows. Ma è anche un'altra libreria da aggiungere alla tua lista e un altro "framework" da imparare, il che conta contro di essa.

Topshelf o .NET Core Workers, fai davvero la tua scelta.

**Post Correlati**

1. [**Creazione di servizi Windows in .NET Core - Parte 2 - Il modo "Topshelf"**](https://dotnetcoretutorials.com/2019/09/27/creating-windows-services-in-net-core-part-2-the-topshelf-way/)
2. [**Creazione di servizi Windows in .NET Core - Parte 1 - Il modo "Microsoft"**](https://dotnetcoretutorials.com/2019/09/19/creating-windows-services-in-net-core-part-1-the-microsoft-way/)
3. [**Hosting di un'app Web ASP.NET Core come servizio Windows in .NET Core 3**](https://dotnetcoretutorials.com/2019/12/21/hosting-an-asp-net-core-web-app-as-a-windows-service-in-net-core-3/)
4. [**Hosting di un'applicazione Web ASP.NET Core come servizio Windows in .NET Core 2**](https://dotnetcoretutorials.com/2018/09/12/hosting-an-asp-net-core-web-application-as-a-windows-service/)