Les Chroniques du Campagnol
Season 2 Episode 5
The main service loop

April 16, 2022

It's time to get started writing some code. Let's start with the beginning, the main program loop.

1 Creating a Windows Service application

The .Net Core platform has native support for writing Windows services. All you have to do is to follow the instructions here.

As usual, I started by doing just what was written, I added some functionality then wondered how all this could be refactored into a generic service loop.

2 BackgroundService

Writing a Windows Service is fairly simple, you subclass the `BackgroundService` class and you implement the `ExecuteAsync` function to provide the service, looping until the service needs to stop. How do you know the service has to stop? Well `ExecuteAsync` receives a `CancellationToken`. When the token gets signalled, you need to exit.

```
1    protected override async Task ExecuteAsync (CancellationToken ServiceStop)
2    {
3        while (!ServiceStop.IsCancellationRequested)
4        {
5            // Do some work
6            await Task.Delay (1000, ServiceStop);
7        }
8    }
```

Strictly speaking, you don't need to check `IsCancellationRequested`, because `Task.Delay` will throw an exception if the cancellation token is signalled. Unfortunately, some calls will throw `TaskCancelledException`, some others will throw the more generic `OperationCancelledException`. Bottomline, if you want to know whether you were asked to exit cleanly, check for the generic exception.

3 Debugging

What is really nice is that the code can run both as a Windows service and as a regular executable. You start it by typing its name (or through `dotnet run`), and stop it with a control-C. That means the exact same code *also* works fine in a Linux terminal.

4 Don't repeat yourself

As you might have guessed, all this is already too cumbersome for me. It means that I will have to pass the cancellation token to every function that I call and optionally check its status from a hard-to-remember property. Let's instead store it in an instance variable and provide meaningful properties.

```
    CancellationToken _Stop;
    public bool Running => !_Stop.IsCancellationRequested;

    public async Task Sleep (int Milliseconds)
    {
        await Task.Delay (Milliseconds, _Stop);
    }

    protected override async Task ExecuteAsync (CancellationToken ServiceStop)
    {
        _Stop = ServiceStop;

        while (Running)
        {
            // Do some work
            await Sleep (1000);
        }
    }
```

5 Computing Modules

More than 20 years ago, when designing LDS V5 (an aborted project, written for Windows 3.1 in 16 bit C++) Raoul had devised the notion of *Computing Modules*. These were inter-dependant "things" that ran inside the executable, each possibly *started*, *stopped*, or *failed*. Or maybe in transient states like *starting* or *stopping*. Yes, exactly like connections in IM...

Each module was able to define its dependencies, so starting a module automatically started all the dependencies. Say you wanted to start the module that computes requests (the equivalent of QMGR) then it automatically started the module that connects to the database.

Computing modules were abandoned for the most part in LPM, as we no longer needed them that much. LPM targeted Windows 32 bits, which was inherently multithreaded, while Windows 16 bits relied on cooperative multitasking. You don't need to think hard to realize that the trend is now back agin to heavily use cooperative multitasking in the form of promises, tasks, coroutines, etc.

Some other time, I'll have to check whether I can make use of the computing modules using promises and tasks. That will also require some refactoring because all this was written several years ago, when .Net Core did not exist, and project files were way more limited.

6 Running parallel tasks

The code is now an order of magnitude more readable. Let's now make it able to run multiple tasks in parallel. After all, that's the goal of asynchronous routines. To do so, let's have a function that creates all the tasks we need to run in parallel, then wait for these to stop. Once all tasks have stopped, we're done.

```
    protected override async Task ExecuteAsync (CancellationToken ServiceStop)
    {
        // ...

        _Tasks.AddRange (StartupTasks ());
        while (_Tasks.Any ())
        {
            Task Completed = await Task.WhenAny (_Tasks);
            if (Completed != null)
            {
                _Tasks.Remove (Completed);
            }
        }
    }
```

Why don't you simply wait on a WhenAll(_Tasks) once instead?

Because one background task could spawn additional tasks. For example, I could have a task that enumerates all the network connections of my machine, then for each of those network connections, create an additional task to explore the other devices attached to that network. Every time a task creates a subtask, it adds it to the `_Tasks` queue and we will wait for it to finish.

The nice thing about this code is that my `ExecuteAsync` function no longer contains any custom code. It only deals with ensuring that the `_Tasks` array gets cleaned of completed tasks so that we perform cleanup when possible.

7 StandardBackgroundService

That's it. I can now have a standard background worker that is completely mission-agnostic and reusable, and my custom class that actually implements the application logic.

```
public abstract class StandardBackgroundService: BackgroundService
{
    protected override async Task ExecuteAsync (CancellationToken ServiceStop)
    {
        // ...
    }
}

public class WorkstationUploaderService : StandardBackgroundService
{
    protected override IEnumerable<Task> StartupTasks ()
    {
        // ...
    }
}
```


8 What's next

Next we will want to actually implement some useful background tasks and instantiate them in the `StartupTasks` function. But that was not as easy as it seems, so we'll first add some helping tools.