Les Chroniques du Campagnol
Season 2 Episode 6
Mercator - Error Management

April 23, 2022

So OK, we need to implement the `StartupTasks` method. That's so easy and silly it does not require a whole article.

Or does it???

Of course not! Let's have a closer look.

1 The initial attempt

That seemed particularly easy as I said. I just defined my initial tasks as `async` functions:

```
    private async Task UploadToApi (LabxApi Api)
    {
        // Whatever
    }

    protected async Task ObserveNetworkInterfaces ()
    {
        // Whatever
    }
```

And called them inside the `StartupTasks` function. Of course I cannot simply do `await` on them. I need to build a `List` with the promises and return it:

```
    protected override IEnumerable<Task> StartupTasks ()
    {
        LabxApi Api = new LabxApi ("http://localhost", "myuser", "mypassword");

        List<Task> t = new List<Task>
        (
            ObserveNetworkInterfaces (),
            UploadToApi (Api)
        );

        int AutoStop = 3600000;
        IniFile.AccessConfigurationMs ("Application", "AutoStop", ref AutoStop);
        if (AutoStop > 0)
        {
            t.Add (StopApplicationIn (AutoStop));
        }

        return t;
    }
```

Urgh! That's ugly and cumbersome. I need to create a list and populate it using `Add` every time there is an optional task. Also, I need to create all needed additional structures (like `Api`[1]) beforehand. Wish there was a way to build a list with optional items, and add ancillary initialisations as you go.

[1] Yes I know that in my example I could do without a local variable, just instantiate the `LabxApi` object as the argument for the function. Actually my code to create the API is a little more convoluted and I spare you the details.

I bet you're going to tell us that there is such a wonder.

Of course there is.

2 A first iteration

I made `StartupTasks` return a `IEnumerable` for a reason. This makes the function a **generator**. C♯ has a fancy syntactic sugar for implementing generator functions: `yield`. So let's use that instead:

```
    protected override IEnumerable<Task> StartupTasks ()
    {
        yield return ObserveNetworkInterfaces ();

        int AutoStop = 3600000;
        IniFile.AccessConfigurationMs ("Application", "AutoStop", ref AutoStop);
        if (AutoStop > 0)
        {
            yield return StopApplicationIn (AutoStop);
        }

        LabxApi = new LabxApi ("http://localhost", "myuser", "mypassword");
        yield return UploadToApi (Api);
    }
```

3 Silence! Action! ... Silence???

I ran the application and... well nothing truly interesting happened in `UploadToApi`. I knew that the first thing the function would to is to log-in to the server and fail miserably (there is no server at the address I mentioned). There should be an error report about that failed attempt; but nothnig could be seen.

It turns out that the function indeed failed to log-in and correctly reported that as an exception. But the log showed absolutely nothing. After some head scratching, I found the core issue: there is no exception reporting at the end of the task, so C♯ duly notes in the promise that an exception occurred, but since no one bothered to ask (nor call `await`), the promise was silently left in its failed state until the program ended, at which point its status became moot and was not reported.

We need to report the error as we remove the completed tasks from the queue

Yes, but alas it's not that simple. Let's try that.

4 Reporting failed tasks

Before I remove them from the queue, I just now `await` them so the exception is reported:

```
    while (_Tasks.Any ())
    {
        Task Completed = await Task.WhenAny (_Tasks);
        if (Completed != null)
        {
            try
            {
                await Completed;
            }
            catch (Exception e)
            {
                Logger.LogError ("Task ended in error {0}", e);
            }
            _Tasks.Remove (Completed);
        }
    }
```

When tasks exit, their errors are reported. But I now have three additional issues. The first is trivial: using `await` to throw an exception and `catch` it immediately is a bit silly. It works but it's silly. Oh, and also I do not filter out the case where the task exits with an `OperationCancelledException`, which is not an error.

The second one is that passing an exception like that will add the full stack trace to the log. It's useful indeed, but remember that the application runs as a Windows service, and the trace will end up in the *Windows Application Event Log*. That tool does not make the slightest attempt at making a stack trace readable.

And the third one is that errors are not reported as they occur, but as bursts when the application stops. I know why, I knew it from the beginning and did not care, but now that I want to report exceptions, I suddenly *do* care.

5 Why errors are sometimes reported as a burst at application stop

This happens because at a given point, the `_Tasks` variable holds *only* long-running tasks (ie daemon tasks that run until the service needs to stop). The `WhenAny` call will therefore only return when one of these daemons stop, and that is only as the application terminates.

If one of these daemons starts another task, and that task fails, it will sit idle in the `_Tasks` queue until one of the daemons exits, and a new `WhenAny` is issued, this time bothering about the other tasks in the queue.

What we need to do is to prematurely return from `WhenAny` each time a task is added to the queue so that we can start a new `WhenAny` with all the tasks in the queue. That seems complicated enough for an entire episode, but the solution to the second problem makes the third less of an issue for the moment, so we'll start by solving that one and bury the other two under the carpet.

6 What's next

Next I will tackle that obnoxious stack trace in the error log. In doing so, I will greatly improve the error reporting mechanism.