
Working with Event-based asynchronous pattern and BackgroundWorker

EAP is a model that has also been introduced to handle threading in an easier and elegant way. The Event-based asynchronous pattern forms few rules that you need to follow while following the pattern. The implementation of Event-based Asynchronous pattern has been widely accepted, which uses events to notify the caller with the changes to the thread.

Getting ready

In this recipe, we are going to show an example of the `BackgroundWorker` type rather than implementing a new EAP class which will use `ThreadPool` in the background and run the method that is passed to it asynchronously. The `BackgroundWorker` class has special features like `ProgressReport`, `CompleteCallback`, or even cancellation of the call, hence it becomes the sole EAP type which most of the developers use to take benefit of.

How to do it...

Let us create a Windows application to show the usage of the `BackgroundWorker` class in the recipe.

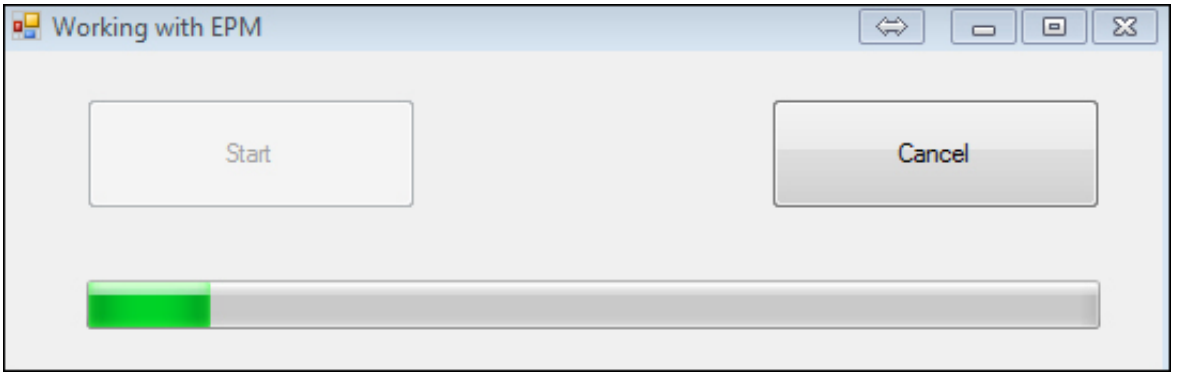
1. Create a method named `XXXAsync` for asynchronous member with the same parameters that are needed to be called for the synchronous member. You can pass a State object as well if needed. This method calls `BeginInvoke` to perform the asynchronous operation.
2. `AsyncCompletedEventArgs`, used as the second parameter, is inherited from `EventArgs` and represents the additional settings that need to be passed to every event when it completes. The `AsyncCompletedEventArgs` parameter itself exposes properties like `Cancelled` (which indicates whether the operation is cancelled), `Error` (holds the error object) or `UserState` (holds the state of the call).
3. Create a delegate with `XXXCompletedEventHandler`, which takes argument as an object, and `XXXCompletedEventArgs`.
4. Create an event of type `XXXCompletedEventHandler`. The event is raised when the asynchronous operation is complete.
5. `BackgroundWorker` is a managed class that has already implemented the EAP. We use `BackgroundWorker` in the following code to implement the asynchrony in the application:

```
BackgroundWorker backgroundWorker;
BackgroundWorker Worker
{
    get
    {
        this.backgroundWorker = this.backgroundWorker ?? new
BackgroundWorker();
        return this.backgroundWorker;
    }
}
public long DoCpuIntensiveWork()
{
    int i = 2, j, rem, result = 0;
    while (i <= 1000000)
    {
        for (j = 2; j < i; j++)
        {
            rem = i % j;
            if (rem == 0)
                break;
        }
```

```
            if (i == j)
                result = i;
            i++;

            int progress = (int)(((float)i /(float)1000000) *
10000);
            if(progress > 0)
                this.Worker.ReportProgress(progress);

            if (this.Worker.CancellationPending)
                return i;
        }
        return result;
}

private void start_Click(object sender, EventArgs e)
{
    this.Worker.RunWorkerAsync();
    this.button1.Enabled = false;
    this.button2.Enabled = true;
}

public void backgroundWorker_DoWork(object sender, EventArgs e)
{
    this.DoCpuIntensiveWork();
}

private void Form1_Load(object sender, EventArgs e)
{
    this.Worker.WorkerSupportsCancellation = true;
    this.Worker.WorkerReportsProgress = true;
    this.Worker.DoWork += new
DoWorkEventHandler(backgroundWorker_DoWork);
    this.Worker.ProgressChanged += new
ProgressChangedEventHandler(Worker_ProgressChanged);
    this.Worker.RunWorkerCompleted += new
RunWorkerCompletedEventHandler(Worker_RunWorkerCompleted);
}

void Worker_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
{
    this.button1.Enabled = true;
    this.button2.Enabled = false;
}

void Worker_ProgressChanged(object sender,
ProgressChangedEventArgs e)
{
    this.progressBar1.Value = e.ProgressPercentage;
}

private void cancel_Click(object sender, EventArgs e)
{
```

```
            this.Worker.CancelAsync();
        }
```

The previous code uses `BackgroundWorker` to implement the functionality of Asynchronous operation. We call `DoCPUIntensiveOpration` from the `DoWork` eventhandler of `BackgroundWorker`. `DoWork` represents the working module. The `ProgressChanged` event is raised when we call the `ReportProgress` method from inside the working module where we need to update the progress bar control on the UI. You should also note that the event `ProgressChanged` automatically detects the right thread from where it is created.



This form displays two buttons which let us start a long running process with the click of the **Start** button and the operation progress is reflected on the progress bar, and the **Cancel** button cancels the operation.

6. When **Cancel** is clicked we request cancellation. The logic inside `DoWork` should check for `CancellationPending`, and depending on its value, it cancels the operation.
7. `RunWorkerCompleted` is called when the operation has finished execution.

How it works...

Event-based Asynchronous pattern works similar to any other asynchronous pattern, but uses events and delegates to implement Call backs. The `BackgroundWorker` class being the classic example of EPM, implements the Event-based Asynchronous pattern. According to this pattern, the class should contain:

- A method with `MethodNameAsync` to invoke the operation.
- There should be a corresponding `MethodNameCompleted` event that will be invoked when method has finished execution. This serves as a call back to the method.

By the way, you should also remember that the `RunWorkerAsync` immediately returns the control, as it runs the method asynchronously. It creates a new Thread from the `ThreadPool` and calls it as a `WorkItem` to the `ThreadPool` thread. It also subscribes a call back that will invoke the `RunWorkerCompleted` event when the method has finished execution.

Additionally, the `BackgroundWorker` class can also report progress of the execution if you call `ReportProgress` from your code, which will invoke the `ProgressChanged` event as we saw before. There are lots of other features of the `BackgroundWorker` class too, and it is recommended to have Event-based Asynchronous Pattern type following .NET threading patterns.

See also

- http://bit.ly/BackgroundWorker
- http://bit.ly/BWUseCases