**Chapter 6: The Threading Patterns**

**Introduction:**

Patterns are tried, tested and recommended way of implementing a solution to for a standard problem, till now we have seen how easy it is to implement an asynchronous method using async/await. In this chapter we will further see in detail on the patterns that are available using async/await and tasks which can be used in implementing enterprise application. We will further investigate exception handling, cancellation, progress tracking in tasks. Further we will touch base on patterns to implement asynchronous methods without async/await and how to implement async/await wrapper methods on legacy methods.

**Structure:**

* Task based Asynchronous Pattern (TAP)
  + Overview
  + Implement TAP
  + CPU Bound vs I/O Bound
  + Exception handling
  + Cancellation
  + Progress reporting
* Asynchronous Programming Model (APM)
  + APM to TAP wrapper
  + TAP to APM wrapper
* Event based Asynchronous Pattern (EAP)
  + EAP to TAP wrapper
* Summary
* Exercise

**Objectives:**

By end of this chapter read should be able to understand

* What are the various threading patterns available in .Net to develop asynchronous applications?
* How to implement each of these patterns in modern application development.
* How to implement wrapper over legacy patterns if need arises.

**Task-based Asynchronous Pattern (TAP)**

Task based Asynchronous Pattern (TAP) is recommended pattern to implement async programming in .NET. Task objects are one of the central components of TAP. This pattern is based on System.Threading.Tasks namespace using Task, Task<T> types or any type that exposes a GetAwaiter() method. In this pattern we create a single method that represent beginning and ending of asynchronous operation.

**Implementing pattern**

To implement this pattern we will start with prefixing function with async keyword and add await keyword to the method that can be performed asynchronously - typically a method retrieving data from database, reading file from disk or an API call (I/O Bound). This is illustrated in below example where we have a button click event on a win form loading data from API synchronously

private void Search\_Click(object sender, EventArgs e)

{

BindingSource bindingSource1 = new BindingSource();

var ticker = new Stopwatch();

ticker.Start();

var request = WebRequest.Create("https://localhost:44394/api/StockSynchronous");

var response = request.GetResponse();

Stream dataStream = response.GetResponseStream();

StreamReader reader = new StreamReader(dataStream);

string responseFromServer = reader.ReadToEnd();

var data = JsonConvert.DeserializeObject<IEnumerable<Stock>>(responseFromServer);

bindingSource1.DataSource = data.Where(price => price.StockName == searchText.Text);

stockData.AutoResizeColumns(DataGridViewAutoSizeColumnsMode.AllCellsExceptHeader);

stockData.DataSource = bindingSource1;

progressMessage.Text = $"Loaded stocks for {searchText.Text} in {ticker.ElapsedMilliseconds}ms";

}

To convert above method to an asynchronous method we prefix it with async and also add asynchronous methods with await for all the outbound calls. The button click event on a win form loading data from API asynchronously will look like below

private async void Search\_Click(object sender, EventArgs e)

{

BindingSource bindingSource1 = new BindingSource();

var ticker = new Stopwatch();

ticker.Start();

using (HttpClient client = new HttpClient())

{

var response = await client.GetAsync($"https://localhost:44394/api/StockS");

var content = await response.Content.ReadAsStringAsync();

var data = JsonConvert.DeserializeObject<IEnumerable<Stock>>(content);

bindingSource1.DataSource = data.Where(price => price.StockName == searchText.Text);

}

stockData.DataSource = bindingSource1;

progressMessage.Text = $"Loaded stocks for {searchText.Text} in {ticker.ElapsedMilliseconds}ms";

}

In the above sample code await keyword helps is getting the result from asynchronous operation once data is available without blocking the UI thread. So, await keyword stores result of the async operation in the left-hand side variable as in this case content variable is a string. The benefit of doing this is that UI thread is returned to the caller and unblocks the UI while data is retrieved from API.

Note – async void is allowed only for UI event handlers, other scenarios it should be avoided we will see on why to avoid later in this chapter

**CPU Bound vs I/O Bound**

When implementing asynchronous code especially on the server side it is important to identify whether method is doing I/0 bound task or CPU Bound task, a simple way to do is to ask whether my method completion is dependent on external sources for example a database call, an API call or load data from a file on disk, async is best fit in such scenarios. However, if you are doing an expensive computational work like executing a business algorithm async is not a best fit as the code will still run synchronously. Let’s see that with an example:

Let us a create a console application and name it CPUBoundvsIOBound, this console application will look like below

using System;

using System.Net.Http;

using System.Threading;

using System.Threading.Tasks;

namespace CPUBoundvsIOBound

{

class Program

{

static void Main()

{

}

}

}

Now we will add 2 methods to this class

GetStocksAsync() – This is s simple async method that makes a call at to an API asynchronously and prints message on success/failure. This method will look like

Async

Add another method DoExpensiveCalculation(), this method will do some in memory calculation asynchronously and consumes significant CPU through a for loop. Code for this method will look like below

Async

Now add another helper method that calculates number of available I/O and CPU threads using System.Threading.ThreadPool class. Definition of this method will look like below

/// <summary>

/// Method to log available threads

/// </summary>

static void AvailableThreads()

{

int worker, io;

ThreadPool.GetAvailableThreads(out worker, out io);

Console.WriteLine("Thread pool threads available at startup: ");

Console.WriteLine(" Worker threads: {0:N0}", worker);

Console.WriteLine(" Asynchronous I/O threads: {0:N0}", io);

}

With this update the main method where we will call both the methods GetStocksAsync() and DoExpensiveCalculationAsync() and print number of available threads using AvailableThreads(). We will use await here for the asynchronous calls hence signature of Main method will change to async Task Main (). With this updated code for Main method will look like below

static async Task Main()

{

Console.WriteLine("Before I/O bound task");

Console.WriteLine("===================================");

AvailableThreads(); //Check number of threads initially

await GetStocksAsync(); //Call async method

Console.WriteLine("After I/O bound task");

Console.WriteLine("===================================");

AvailableThreads(); //Check number of threads

await DoExpensiveCalculationAsync(); //Call a method that does CPU intense operation

Console.WriteLine("After CPU bound task");

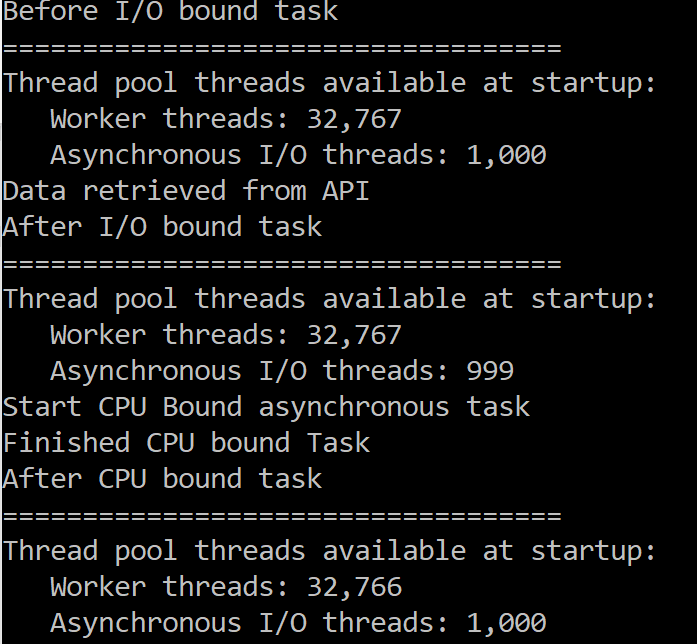
Console.WriteLine("===================================");

AvailableThreads(); //Check number of threads

Console.ReadLine();

}

Running the code above will give output as shown in Figure 6.1,



**Figure 6.1 – Output of threads used for CPU bound vs I/O bound tasks**

Here you can see that although we are calling a background thread for CPU intensive operation it has used a worker thread which is ok for client side application for things like unblocking UI, However for an ASP.Net application this is will not lead any significant gain and also lead to possible deadlock as this is nothing different than running operation synchronously as there is no added benefit of assigning a dedicated thread for CPU bound operation.

So to conclude on I/O vs CPU bound tasks let us Take an analogy of buying tickets at movie counter (assuming this is the only way to book tickets)

* You can tell your friend to buy popcorn while you are waiting in queue.
* However, to buy tickets there is no alternative but to reach counter, even assuming multiple counter scenario (multiple core) where number of people ahead of you are same across counters, switching across counters is not going to save any additional time.

Understanding CPU bound vs I/O bound task is a key deciding factor on how we can gain benefit of asynchronous programming, as making CPU bound method asynchronous is not going to boost the performance of the application significantly where as making an I/O call is going to give us a significant overall efficiency in the application.

Note – In reality there Is no thread dedicated for I/O operations because we do not need dedicated CPU time, as time spent is primarily receiving data over network or reading data from disk so your application will use available resources more effectively leading to a better responsiveness faster loading time etc.

**Exception Handling**

Exception handling in async methods based on TAP pattern is nothing different than exception handling in any other method in C# i.e. add a try-catch-finally block to your code and you are good to go. Although this is over simplification of exception handling this is the benefit of using async-await keywords for your asynchronous operations, here compiler is taking care of chaining exception back to the caller and unwrapped exception is thrown back to the calling method. Let us see this with simple example as below where we create an asynchronous method calling an API and is throwing exception and will have a caller method where we doing basic exception handling

using System.Collections.Generic;

using System.Net.Http;

using System.Threading.Tasks;

namespace TAPExceptionHandling

{

class Program

{

static async Task Main(string[] args)

{

var task = GetDataAsync();

try

{

var data = await task;

Console.WriteLine(data);

}

catch (Exception ex)

{

Console.WriteLine($"Exception occured in GetDataAsync method - {ex.Message} \n Innerstack \n {ex.StackTrace}");

}

Console.Read();

}

/// <summary>

/// Async method to retrieve data from API

/// </summary>

/// <returns></returns>

static async Task<string> GetDataAsync()

{

using (HttpClient client = new HttpClient())

{

try

{

var response = await client.GetAsync("https://localhost:44394/api"); // Giving a non-existing API method to generate exception

response.EnsureSuccessStatusCode();

var content = await response.Content.ReadAsStringAsync();

Console.WriteLine($"Data retrieved from API");

return content;

}

catch

{

throw;

}

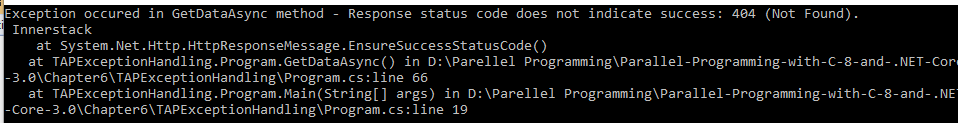
}

}

}

}

Once we run the above code we will see output as shown in Figure 6.2



**Figure 6.2 – Basic exception handling**

Another way to retrieve exception is to read the exception property of Task variable, in above example it’s task.Exception. So, let us add below code to the catch block of main method

List<String> errors = task.Exception.Flatten().InnerExceptions.Select(x => x.Message).ToList();

int counter = 0;

foreach (string error in errors)

{

counter++;

Console.WriteLine($"{counter}).Error - {error}");

}

Once we run this code we will see same output as shown in Figure 6.2 i.e. inner stack of the exception getting printed just like with the previous code. So, in this sample we have seen a simple way to handle exceptions in asynchronous code in next sections we will see how to handle exceptions in more complex scenarios.

Note – Calling async method without await is not going to propagate exception and will be swallowed.

**Nested Exception Handling**

In real scenario there would be more than one asynchronous calls/multiple tasks, handling that wouldn’t be any different i.e. use await and original exception is unwrapped and is propagated to the caller. This is illustrated in below example where we create a simple console application let us call it TAPExceptionHandling. This application will have 3 methods out of which first method will be a method getting data from API asynchronously, adding that to this console application code will look like below

using System;

using System.Collections.Generic;

using System.IO;

using System.Net.Http;

using System.Threading.Tasks;

using System.Linq;

namespace TAPExceptionHandling

{

class Program

{

static void Main(string[] args)

{

}

/// <summary>

/// Async method to retrieve data from API

/// </summary>

/// <returns></returns>

static async Task<string> GetDataAsync()

{

using (HttpClient client = new HttpClient())

{

try

{

var response = await client.GetAsync("https://localhost:44394/api"); // Giving a non existing API method to generate exception

response.EnsureSuccessStatusCode();

var content = await response.Content.ReadAsStringAsync();

Console.WriteLine($"Data retrieved from API");

return content;

}

catch

{

throw;

}

}

}

}

}

Now add another helper method that will just call GetDataSync(), let’s call this GetDataSyncNested(). The purpose of this call is to simulate hierarchy in function call to simulating nesting.

Now create another method which retrieves data from a file in this case a non-existing file to so that it can throw an exception, this method will look like below

/// <summary>

/// Async method to retrieve data from API

/// </summary>

/// <returns></returns>

static async Task<string> GetDataAsyncFromAnotherSource()

{

try

{

using (var stream = new StreamReader(File.OpenRead(@"nonexistingfile.txt")))

{

var fileText = await stream.ReadToEndAsync();

Console.WriteLine("Reading from file completed");

return fileText;

}

}

catch

{

throw;

}

}

With all this let us update main method where we call both these methods i.e. nested one calling API and another method calling non existing file. This will simulate a scenario of nested and multiple exceptions and the way we need to handle it is by adding a try catch block across the tasks and in the catch loop through to write it to the console, with that main method will look like below

static async Task Main(string[] args)

{

var taskfromAPI = GetDataAsyncNested();

var taskFromFile = GetDataAsyncFromAnotherSource();

var tasks = new List<Task<string>>();

tasks.Add(taskfromAPI);

tasks.Add(taskFromFile);

var allTasks = Task.WhenAll(tasks);

try

{

await allTasks;

}

catch

{

List<Tuple<string, string>> errors = allTasks.Exception.Flatten().InnerExceptions.Select(x => new Tuple<string,string>(x.Message, x.StackTrace)).ToList();

int counter = 0;

foreach (Tuple<string, string> error in errors)

{

counter++;

Console.WriteLine($"{counter}).Error - {error.Item1} \n Innerstack \n {error.Item2} \n");

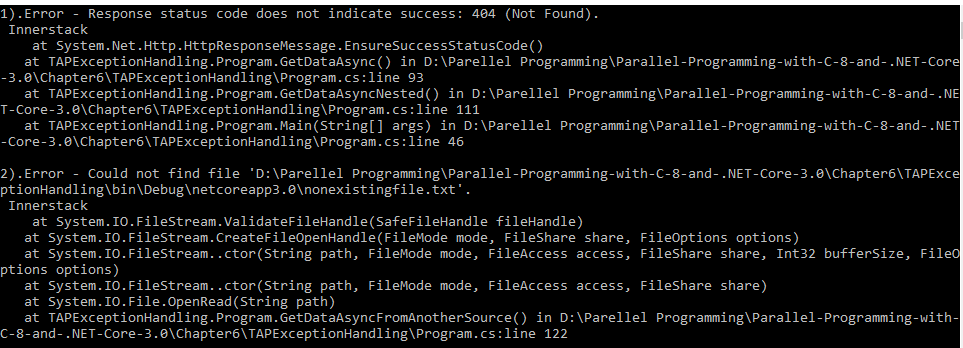
}

}

Console.Read();

}

Once we run this code output will look like as shown in Figure 6.3 where we can see multiple exceptions from asynchronous methods handled and printed on the console (off course in enterprise application we will do more than printing it to console.)



**Figure 6.3 – Nested exception handling**

So, this way we can easily handle multiple or nested exceptions by simply wrapping the asynchronous calls in try catch block and catching the exception.

**Exception Handling in Task.Wait()**

However, things are different when await keyword is not used for an async operation and is implemented using task.wait(). All the exceptions are wrapped in AggregateException and thrown to the calling thread. Calling method can specifically catch AggregateException to loop through and act accordingly. Let us change main method in previous example a bit as shown below, here instead of await we are going to use Wait() method of Task for all the task completion and because of which now exceptions are also available in AggregateException . With this our main method will look like below

static void Main(string[] args)

{

var taskfromAPI = GetDataAsyncNested();

var taskFromFile = GetDataAsyncFromAnotherSource();

var tasks = new List<Task<string>>();

tasks.Add(taskfromAPI);

tasks.Add(taskFromFile);

var allTasks = Task.WhenAll(tasks);

try

{

//await allTasks;

Task.WhenAll(allTasks).Wait();

}

catch (AggregateException agex)

{

List<Tuple<string, string>> errors = agex.Flatten().InnerExceptions.Select(x => new Tuple<string, string>(x.Message, x.StackTrace)).ToList();

int counter = 0;

foreach (Tuple<string, string> error in errors)

{

counter++;

Console.WriteLine($"{counter}).Error - {error.Item1} \n Innerstack \n {error.Item2} \n");

}

}

}

Once we run this code output of this code is the same as the one in Figure 6.3. So, this way we can additionally catch AggregateException and handle it accordingly.

**Using handle method**

There could be scenarios where we do not want to propagate specific type of exceptions to parent and do some action in child method itself. In such cases AggregateException gives handle method to filter exceptions and act accordingly, input to it is a function delegate which will be called for each exception and it needs to be handled with in async method we need to return true, else return false. In short handle method is for “Handled” exceptions and all unhandled exceptions can be propagated to calling method, following example illustrates that behavior. We will use the same console application in previous example and additionally add another asynchronous method where we handle specific exception and pass custom message in case of that exception, this method will loop through directory and throw an exception which will be handled and thrown to caller with a custom message. Let’s call this method DoHoghCPUIntense() and definition of this method will look like below -

private static string DoHighCPUIntense()

{

String location = @"C:\";

Task<string> output = Task.Run(() =>

{

List<string> files = new List<string>();

for (int i = 0; i < 5; i++)

{

files.AddRange(Directory.GetFiles(location, "\*.txt", SearchOption.AllDirectories).ToList());

}

return files.FirstOrDefault();

});

try

{

output.Wait();

}

catch (AggregateException agEx)

{

//Further handle method can be used to do specific action based on the type of exception

agEx.Handle(x =>

{

if (x is UnauthorizedAccessException)

{

Console.WriteLine("Specific action for UnauthorizedAccessException");

}

return true;

});

}

return string.Empty;

}

Now modify the main method to call this new method along with GetDataAsync method and catch AggregateException, however this time output will have custom message. Main method will look like below now

static void Main(string[] args)

{

var tasks = new List<Task>();

var task = GetDataAsync();

tasks.Add(task);

var task2 = Task.Run(() => DoHighCPUIntense());

tasks.Add(task2);

try

{

Task.WhenAll(tasks).Wait();

}

catch (AggregateException agEx)

{

List<Tuple<string, string>> errors = agEx.Flatten().InnerExceptions.Select(x => new Tuple<string, string>(x.Message, x.StackTrace)).ToList();

int counter = 0;

foreach (Tuple<string, string> error in errors)

{

counter++;

Console.WriteLine($"{counter}).Error - {error.Item1} \n Innerstack \n {error.Item2} \n");

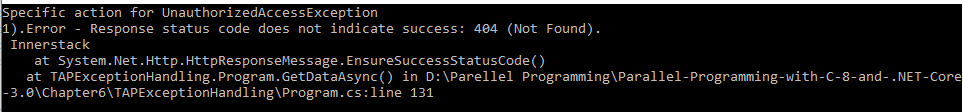
}

}

Console.Read();

}

Here is output in Figure 6.4 for this method and we can see child method hasn’t propagated UnauthorizedAccessException like earlier as it is “Handled” now.



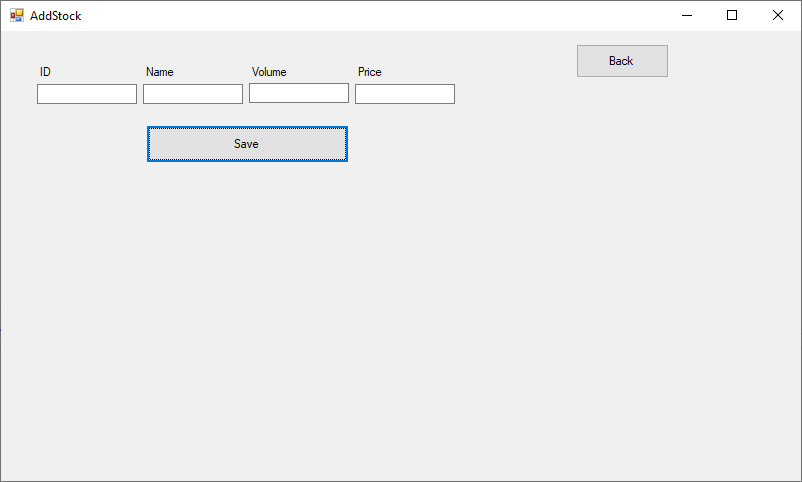
**Figure 6.4 – Exception handling using handle method**

This way further control the exception that is propagated to the caller method and this is very useful if we are building libraries or APIs consumed by third party developers as it helps troubleshoot/debug much easily from caller side.

**Avoid async void**

If you have noticed in all examples we discussed till now there are methods which aren’t returning any data (not even a success flag) however we still have method signature as async Task. The reason behind that is, one of the advantage of returning Task is that entire async operation is represented by Task object which in turn also can be used to identify success or failure of async operation. Further to it any failure can be drilled down based on the exceptions that are placed on the task object and can be handled accordingly. For async Void methods as there is no Task object any exception raised would be raised by generic exception handler and can lead irregular behavior like application crash or w3wp crash in case of web application.

This is true even in case of fire and forget kind of operations like saving data to a central logging data store which is not very critical for user flows and can be async operation still it is recommended to avoid using async void as method signature, instead use async Task for better maintainability of the application and definitely avoid unexpected crashes in the application. Let’s create a simple winform application, add a form and 6 controls (4 text boxes and 2 buttons. The form should look as shown in Figure 6.5.



**Figure 6.5 – Windows form to save data through an API**

Add a private helper method to this form that sends data to an API and returns message of successful save or failure. Let’s have the return type of this method as async void, this helper method will look like below

Now add click event handler for the save button and in this event of save button we need to make a fire and forget API call to save data using private helper method created before. Following code shows that

/// <summary>

/// Button click event

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void SaveStock\_Click(object sender, EventArgs e)

{

errorMessage.Text = "";

try

{

SaveDataAsyncVoid();

}

catch (Exception ex)

{

//This is never caught

errorMessage.Text = $"Exception occurred in SaveDataAsyncVoid method - {ex.Message} \n Innerstack \n {ex.StackTrace}";

}

finally

{

setID.Text = "";

setStockName.Text = "";

setStockPrice.Text = "";

setStockVolume.Text = "";

}

}

Once you click save if there is an exception from API you can see that app crashes and that’s because an exception is raised on the SynchronizationContext of the UI thread. So to handle this either we need build custom SynchronizationContext or easier change method signature to private async Task SaveDataAsyncVoid() and add await in the button click while calling SaveDataAsyncVoid and it’s signature to private async void SaveStock\_Click(object sender, EventArgs e).

Only exception to this would be event handlers as event handler are never called explicitly i.e. caller of event handler is not directly interested on the response. [Todo – Add more reasoning why async void is ok for event handler]

**Cancellation**

One of the advantages with TAP is the ease with which asynchronous operation can be cancelled. Cancelling an operation plays a big role in enhancing user experience and gives flexibility to users on cancelling, For example, consider a search operation in a form with a typo, think about the experience when a user needs to wait for the operation to complete although user is aware that the search result is going to be incorrect because of the typo.

So when implementing any async method using TAP additionally a cancellation token can be passed that can be used to cancel async operation and return to the calling method. Cancellation operation throws an exception of type OperationCanceledException so calling method needs to handle cancelled async operation gracefully. This not only gives better user experience but also frees any thread occupied by I/O bound operation, in case of client side it frees any CPU resources used.

Below code illustrates this where we reuse a win forms application that search stocks from our API, however a cancellation is needed while search isn’t completed:

CancellationTokenSource cts = null;

/// <summary>

/// Search stock click event handler

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private async void Search\_Click(object sender, EventArgs e)

{

stockData.Rows.Clear();

stockData.Refresh();

var ticker = new Stopwatch();

ticker.Start();

search.Text = "Cancel";

//On clicking of Search/Cancel checking to cancel operation or perform search

if (cts != null)

{

cts.Cancel();

cts = null;

return;

}

this.cts = new CancellationTokenSource();

//Delegate on cancellation token when there is a cancellation, executes on calling thread's context in this case UI

this.cts.Token.Register(() =>

{

progressMessage.Text = "Search is cancelled" ;

});

//Cancellation needs to be handled gracefully

try

{

var getData = await GetDataFromAPIAsync(searchText.Text, this.cts.Token);

stockData.DataSource = getData;

}

catch (OperationCanceledException ex)

{

Logs.Text = ex.Message;

}

finally

{

cts = null;

}

progressMessage.Text = $"Loaded stocks for {searchText.Text} in {ticker.ElapsedMilliseconds}ms";

search.Text = "Search";

}

/// <summary>

/// Async method to retrieve data from stocks API

/// </summary>

/// <param name="intputSearchtext">Search text</param>

/// <param name="ctsAPI">Cancellation token</param>

/// <returns>Binding source</returns>

private async Task<BindingSource> GetDataFromAPIAsync(string intputSearchtext, CancellationToken ctsAPI)

{

BindingSource bindingSource1 = new BindingSource();

Uri requestUri = new Uri("https://localhost:44394/api/Stocks");

using (HttpClient client = new HttpClient())

{

var response = await client.GetAsync(requestUri, ctsAPI);

response.EnsureSuccessStatusCode();

var content = await response.Content.ReadAsStringAsync();

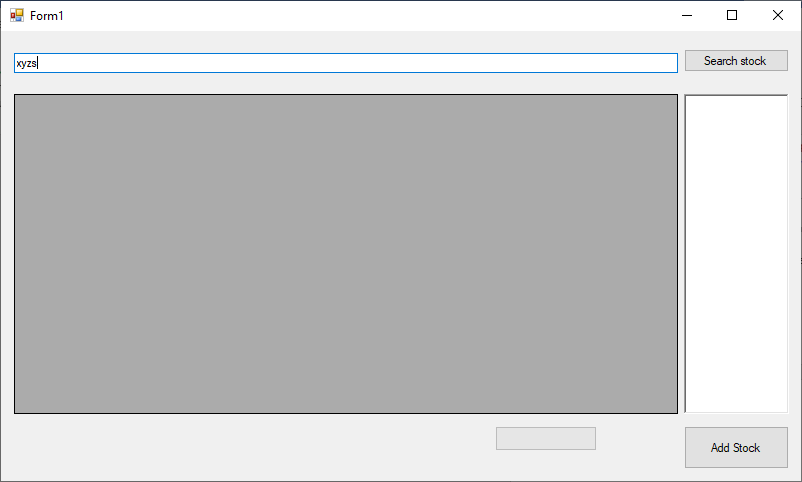
var data = JsonConvert.DeserializeObject<IEnumerable<Stock>>(content);

bindingSource1.DataSource = data.Where(price => price.StockName == intputSearchtext);

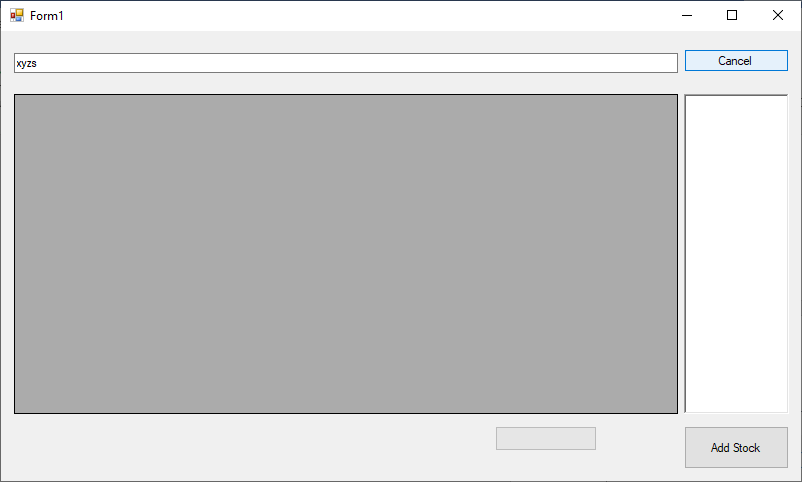
}

return bindingSource1;

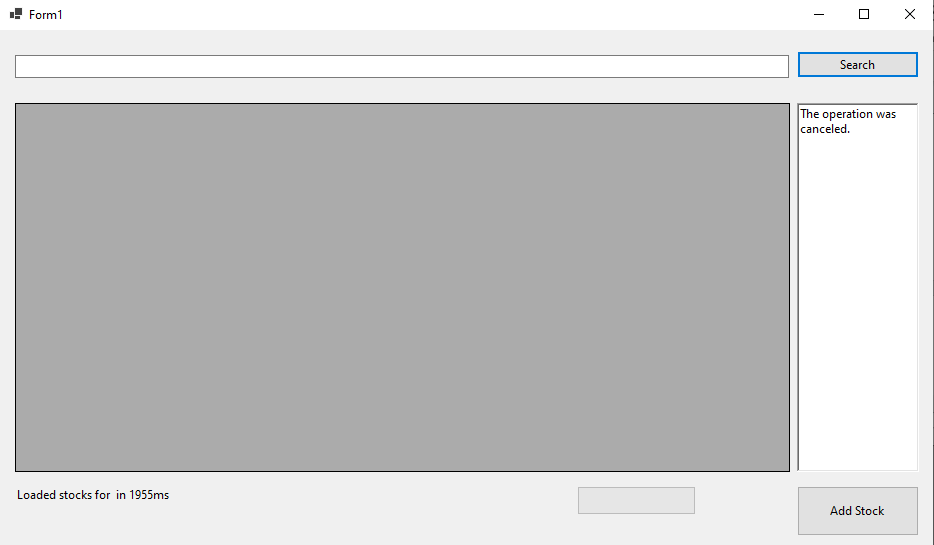
}



**Figure 6.6 – Windows form to search stock – “xyzs”**



**Figure 6.7 – Windows form showing cancel button**



**Figure 6.8 – Windows form after cancelling search operation**

In the above example user tried to search stock “xyzs” however cancelled operation immediately and that has returned from async operation with OperationCanceledException and grid is never loaded. This gives better user experiences as it allows user to search again if needed. Cancellation token also gives us option to subscribe a call back which again runs on calling thread to perform any specific operation. In this example it is used to update progress text as illustrated in the code below

//Delegate on cancellation token when there is a cancellation, executes on calling thread's context in this case UI

this.cts.Token.Register(() =>

{

progressMessage.Text = "Search is cancelled" ;

});

With .net core cancellation token’s callback method would be the first code that is executed after cancellation, so in our code progressMessage.Text is updated final output is “Loaded stocks for in…” and that’s because as soon as cancellation is triggered remaining of the caller method’s code is executed after executing callback. **[Todo – Validate this]**

Note – With .net framework this was the other way around, i.e. first caller’s remaining code is executed and then the callback is executed.

However, there could be cases where there is need that async operation doesn’t throw exception (OperationCanceledException) but return normally. However, in this case calling method need not handle such exception For example Calculating prime numbers less than a huge number or reading line by line from a file, in such cases there may be need to use partial data that is received/processed. A similar example is shown below and its output is shown in Figure 6.9

/// <summary>

/// Async method doing high CPU operation, Add this to form

/// </summary>

/// <returns></returns>

private async Task<long> DoHighCPUIntense(CancellationToken token)

{

long counter = 0;

search.Text = "Stop";

Task<long> output = Task.Run(() =>

{

while (true)

{

counter++;

if (token.IsCancellationRequested)

{

counter++;

break;

}

}

return counter;

}, token);

try

{

await output;

}

catch (AggregateException agEx)

{

throw agEx;

}

return counter;

}

Further call this method in search button click

# region "Scenario 1"

//stockData.DataSource = await GetDataFromAPIAsync(searchText.Text, cts.Token);

//Logs.Text += "API returned data" + Environment.NewLine;

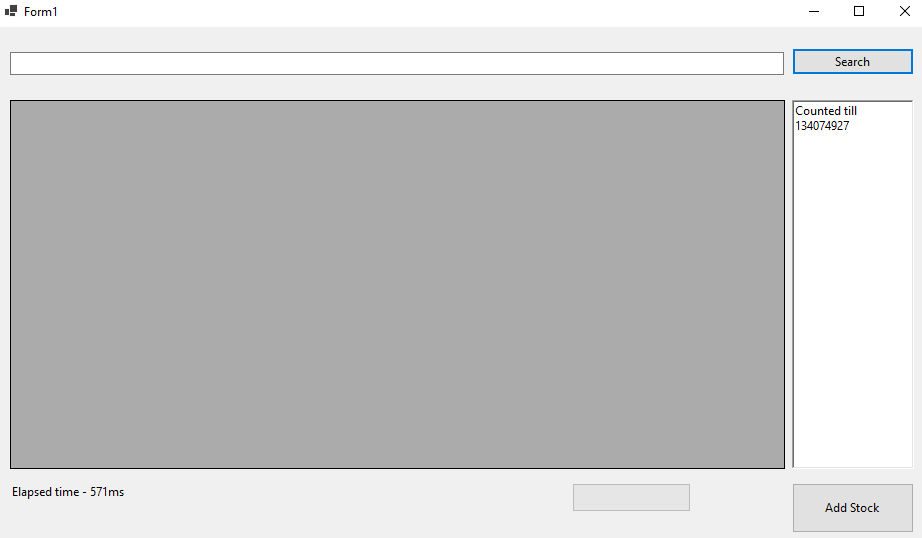
#endregion

# region "Scenario 2"

highCPUCount = await DoHighCPUIntense(cts.Token);

Logs.Text += $"Counted till {highCPUCount.ToString()}" + Environment.NewLine;

#endregion



**Figure 6.9 – Windows form after cancelling search operation and handling without exception**

Cancellation in this method is validating if cancellation is requested and then returning gracefully to caller which further handles. You can notice that in the second example code won’t go to the catch block.

Another important feature of cancellation token is that it allows to pass CancellationToken.None to any method that accepts cancellation token, this indicates that calling method can never cancel this method. For all the framework methods there will be an overload available that accepts cancellation token, similar overload can be provided by developers if they are building libraries/APIs that support async and are developed using TAP.

**Progress reporting**

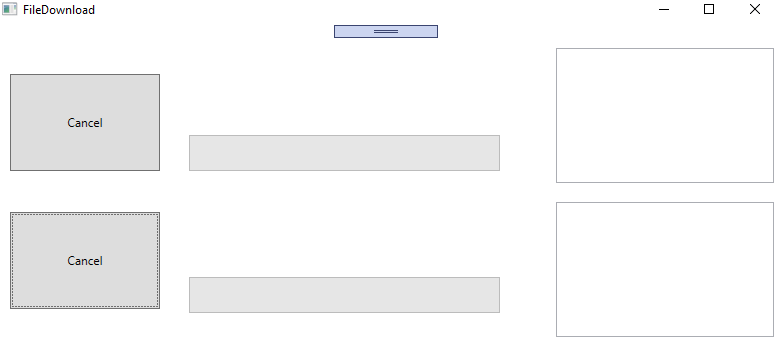
With async pattern developer has the capability to run multiple long running operations in background, reporting progress on how much of operation is completed/pending is important feature to enhance user experience, this helps users to cancel a long running request and gives more fluidic behavior to your application. Typically, applications will have this a progress bar and a cancel button to indicate the amount of work done/remaining and cancel button if user wants to opt out from a long running operation.

With async methods this can be achieved by adding additional parameter of type IProgress<T> to the signature of async method, here T can be any type that will hold progress information.

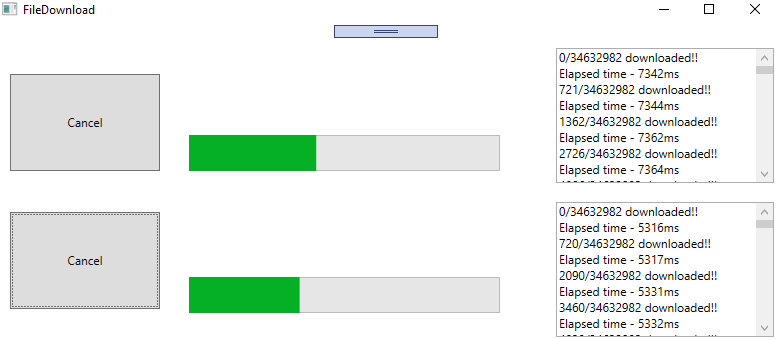
For example, it can be simple integer type where amount of work completed can be reported back in percentage and serves purpose for most of the scenarios or a complex type that holds additional information like time taken between progressing from say 10% to 20% (Will see this further in our example).

IProgress provides Report(T) method to pass progress information back to the UI thread from a background task. This information can be used by say a progressbar control in WPF app to update user on the completed/remaining work.

Let’s consider a simple WPF application that downloads multiple files from web and write it to disk locally.



**Figure 6.10 – Fine download WPF application**



**Figure 6.11 – Fine download WPF application, reporting progress**

As mentioned, the first thing we need to do is add an additional parameter to the download method of type IProgress<T>, in this case method signature supporting progress will look something like below.

private async Task DownloadLargeFilAsync(string fileToDownload, string fileName, CancellationToken token, IProgress<ProgreesReport> progress)

Here T is custom type as we want to pass additional information to the progressbar, progress is reported back by calling report method

progress.Report(new ProgreesReport()

{

progressPercentage = (bytes \* 1d) / (totalLength \* 1d) \* 100,

bytesToRead = bytes,

totalBytes = totalLength,

elapsedTime = ticker.ElapsedMilliseconds

});

In calling method i.e. button click in this case, we create object of Type Progress<T> where T is of type ProgressReport and pass it to async method which takes care of invoking every time progress.Report(T) is called. Progress<T> is a framework class that implements IProgress<T>, when we are creating object of this class it saves current thread’s (in this case UI thread) SynchronizationContext and each time report is called it raises event handlers on the calling thread. As you can see there are 2 ways to do it – either by constructor or through ProgressChanged event.

//Progres reporting

var progress = new Progress<ProgreesReport>(percent =>

{

taskProgress.Value = percent.progressPercentage;

logs.Text += $"{percent.bytesToRead}/{percent.totalBytes} downloaded!!{Environment.NewLine}";

logs.Text += $"Elapsed time - {percent.elapsedTime}ms{Environment.NewLine}";

});

OR

var progress = new Progress<ProgreesReport>();

progress.ProgressChanged += (s, e) =>

{

taskProgress.Value = e.progressPercentage;

logs.Text += $"{e.bytesToRead}/{e.totalBytes} downloaded!!{Environment.NewLine}";

logs.Text += $"Elapsed time - {e.elapsedTime}ms{Environment.NewLine}";

};

Complete implementation of WPF app is as a below

namespace Stocks.WPF

{

/// <summary>

/// Interaction logic for FileDownload.xaml

/// </summary>

public partial class FileDownload : Window

{

CancellationTokenSource cts = null;

CancellationTokenSource cts1 = null;

const string fileName = "largefile.zip";

/// <summary>

/// Button Click

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private async void FileDownload\_Click(object sender, RoutedEventArgs e)

{

fileDownload.Content = "Cancel";

taskProgress.Visibility = Visibility.Visible;

taskProgress.IsIndeterminate = false;

taskProgress.Value = 0;

taskProgress.Maximum = 100;

logs.Text = "";

//On clicking of Search/Cancel checking to cancel operation or perform search

if (cts != null)

{

cts.Cancel();

cts = null;

return;

}

cts = new CancellationTokenSource();

//Progres reporting

var progress = new Progress<ProgreesReport>(percent =>

{

taskProgress.Value = percent.progressPercentage;

logs.Text += $"{percent.bytesToRead}/{percent.totalBytes} downloaded!!{Environment.NewLine}";

logs.Text += $"Elapsed time - {percent.elapsedTime}ms{Environment.NewLine}";

});

try

{

await DownloadLargeFilAsync("https://github.com/Ravindra-a/largefile/archive/master.zip", "largefile.zip", cts.Token, progress);

logs.Text += $"File {fileName} downloaded successfully!!{Environment.NewLine}";

}

catch (OperationCanceledException ex)

{

logs.Text = ex.Message;

}

catch (Exception ex)

{

logs.Text = ex.Message;

}

finally

{

cts = null;

taskProgress.Visibility = Visibility.Hidden;

fileDownload.Content = "File Download";

totalTimeTaken.Content = "Download largefile.zip completed";

}

}

/// <summary>

/// Async Download Method

/// </summary>

/// <param name="fileToDownload">File to download</param>

/// <param name="fileName">Name of file to write locally</param>

/// <param name="token">Cancellation token</param>

/// <param name="progress">Progress reporting</param>

/// <returns></returns>

private async Task DownloadLargeFilAsync(string fileToDownload, string fileName, CancellationToken token, IProgress<ProgreesReport> progress = null)

{

var ticker = new Stopwatch();

ticker.Start();

byte[] buffer = new byte[8192];

int bytes = 0;

string fileToWriteTo = System.IO.Path.Combine(System.IO.Path.GetTempPath(), fileName);

using (HttpClient client = new HttpClient())

{

string url = fileToDownload;

using (HttpResponseMessage response = await client.GetAsync(url, HttpCompletionOption.ResponseHeadersRead, token))

{

response.EnsureSuccessStatusCode();

long totalLength = response.Content.Headers.ContentLength.HasValue ? response.Content.Headers.ContentLength.Value : 34632982; //Once in a while github returns response without content length header

//hence in that case defaulting to actual file size

using (Stream stream = await response.Content.ReadAsStreamAsync(), fileStreamToWrite = new FileStream(fileToWriteTo, FileMode.Create, FileAccess.Write, FileShare.None, 1024, true))

{

for (; ; )

{

int dataToRead = await stream.ReadAsync(buffer, 0, buffer.Length, token);

if (dataToRead == 0)

{

break;

}

else

{

await fileStreamToWrite.WriteAsync(buffer, 0, dataToRead); //Writing stream to disk as and when chunk is downloaded

var data = new byte[dataToRead];

buffer.ToList().CopyTo(0, data, 0, dataToRead);

bytes += dataToRead;

if (progress != null) //For calling methods that do no want to report progress

{

if (((bytes \* 100)/ totalLength ) % 5 == 0) //reporting progress for every 5%

{

progress.Report(new ProgreesReport()

{

progressPercentage = (bytes \* 1d) / (totalLength \* 1d) \* 100,

bytesToRead = bytes,

totalBytes = totalLength,

elapsedTime = ticker.ElapsedMilliseconds

});

}

}

}

}

}

}

}

}

/// <summary>

/// Button click

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private async void FileDownload1\_Click(object sender, RoutedEventArgs e)

{

fileDownload1.Content = "Cancel";

taskProgress1.Visibility = Visibility.Visible;

taskProgress1.IsIndeterminate = false;

taskProgress1.Value = 0;

taskProgress1.Maximum = 100;

logs1.Text = "";

//On clicking of Search/Cancel checking to cancel operation or perform search

if (cts1 != null)

{

cts1.Cancel();

cts1 = null;

return;

}

cts1 = new CancellationTokenSource();

//Progress reporting

var progress = new Progress<ProgreesReport>(percent =>

{

taskProgress1.Value = percent.progressPercentage;

logs1.Text += $"{percent.bytesToRead}/{percent.totalBytes} downloaded!!{Environment.NewLine}";

logs1.Text += $"Elapsed time - {percent.elapsedTime}ms{Environment.NewLine}";

});

try

{

await DownloadLargeFilAsync("https://github.com/Ravindra-a/largefile/archive/master.zip", "largefile1.zip", cts1.Token, progress);

logs1.Text += $"File {fileName} downloaded successfully!!{Environment.NewLine}";

}

catch (OperationCanceledException ex)

{

logs1.Text = ex.Message;

}

catch (Exception ex)

{

logs1.Text = ex.Message;

}

finally

{

cts1 = null;

taskProgress1.Visibility = Visibility.Hidden;

fileDownload1.Content = "File Download";

totalTimeTaken1.Content = "Download largefile1.zip completed";

}

}

}

public class ProgreesReport

{

public double progressPercentage { get; set; }

public long totalBytes { get; set; }

public int bytesToRead { get; set; }

public long elapsedTime { get; set; }

}

}

As you can see in the example this works well in tandem with cancellation it is recommended to implement cancellation along with progress to provide seamless experience for end users. Here we saw on how easily we are able to build an application using Task based async pattern that gives a better user experience by reporting on progress using Progress<T>, IProgress<T>. These principles can further be used to build much more real-world complex applications.

Note – IProgress<T> is not exclusive for asynchronous methods, it can be very well used in synchronous methods.

**Other Asynchronous Patterns**

Till now we have seen on how to implement asynchronous pattern using aysnc/await, however asynchronous methods existing even before async/await. In the next section patterns on how asynchronous methods were implement before async/await and how to build wrappers on top of them so that they can consumed using async/await in any modern application.

**Asynchronous Programming Model (APM)**

Asynchronous Programming Model (APM) also known as IAsyncresult pattern is one of the legacy patterns using which asynchronous operations can be implemented. This pattern expects asynchronous operation to be split into 2 methods one starting with Begin and another starting with End, something like BeginRead and EndRead and an optional callback method.

* Begin method is used to start asynchronous operation where return type of such an operation should be of type IAsyncresult.
* End method takes a parameter of type IAsyncresult i.e. output of Begin method. This used to indicate completion of async operation and to retrieve result/output of asynchronous operation
* An optional callback is passed which gets triggered on completion of begin operation, typically this is used to call End method

Taking example of a typical TAP method from framework Stream class

public Task WriteAsync(byte[] buffer, int offset, int count)

Corresponding APM methods look like below

public virtual IAsyncResult BeginWrite(byte[] buffer, int offset, int count, AsyncCallback callback, object state)

public virtual void EndWrite(IAsyncResult asyncResult)

So, a simple file read asynchronous operation representing a file read will look like below

class Program

{

static Byte[] bytes = new Byte[100];

static void Main(string[] args)

{

Stopwatch watch = new Stopwatch();

watch.Start();

FileStream fs = new FileStream(@"../../../TextFile.txt", FileMode.Open, FileAccess.Read, FileShare.Read, bytes.Length, FileOptions.Asynchronous);

Console.WriteLine($"Begin reading file, Elapsed time - {watch.ElapsedMilliseconds}");

IAsyncResult result = fs.BeginRead(bytes, 0, bytes.Length, null, null);

while (!result.IsCompleted) // Proceeding with doing some other operation while file is being read

{

Console.WriteLine($"Do something else in main method while reading file, Elapsed time - {watch.ElapsedMilliseconds}");

}

int numBytesRead = fs.EndRead(result);

Console.WriteLine($"End reading file, Number of bytes - {numBytesRead}, Elapsed time - {watch.ElapsedMilliseconds}");

fs.Close();

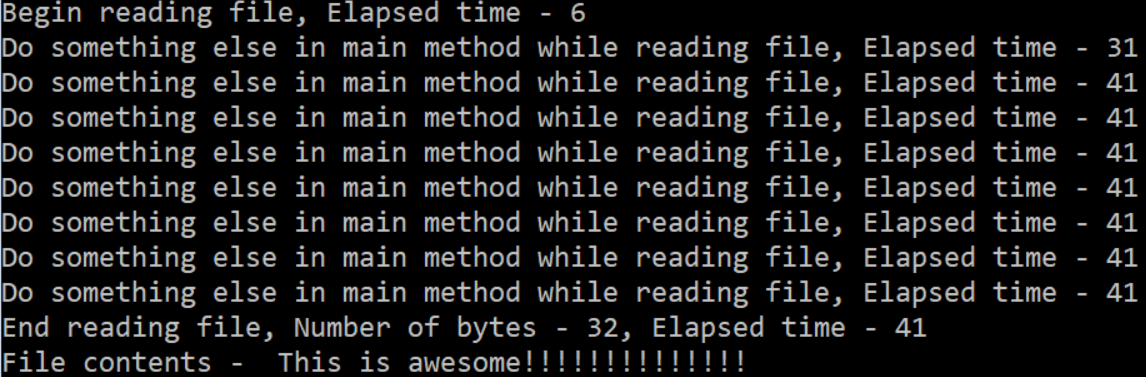
watch.Stop();

Console.WriteLine($"File contents - {Encoding.Default.GetString(bytes)}");

Console.ReadKey();

}

}



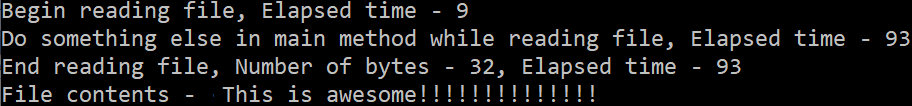
**Figure 6.12 – Output of file read using APM pattern**

Above operation synchronously

int numBytesRead = fs.Read(bytes, 0, bytes.Length);

Console.WriteLine($"Do something else in main method while reading file, Elapsed time - {watch.ElapsedMilliseconds}");

Console.WriteLine($"End reading file, Number of bytes - {numBytesRead}, Elapsed time - {watch.ElapsedMilliseconds}");

****

**Figure 6.13 – Output of file reading synchronously**

Here we can clearly see that in Figure 6.13 line with message “Do Something else..” is executed only after read operation is completed because there we are reading file synchronously however, with APM (Fig 6.12)s way we can parallelly do something else while file is being red.

However, calling EndRead immediately after BeginRead won’t be a realistic scenario that’s where Begin method of APM need to support optional callback operation which gets called once asynchronous operation is completed.

Changing same example above with a callback would look like below, additionally here we can see that the callback operation is performed on a different thread as intended.

class Program

{

static Byte[] bytes = new Byte[100];

static void Main(string[] args)

{

Console.WriteLine($" Managed Thread Id in Main is : {Thread.CurrentThread.ManagedThreadId}"); //// The managed thread identifier.

Stopwatch watch = new Stopwatch();

watch.Start();

FileStream fs = new FileStream(@"../../../TextFile.txt", FileMode.Open, FileAccess.Read, FileShare.Read, bytes.Length, FileOptions.Asynchronous);

Console.WriteLine($" Begin reading file, Elapsed time - {watch.ElapsedMilliseconds}");

fs.BeginRead(bytes, 0, bytes.Length, EndRead, fs);

Console.WriteLine($" Do something else in main method while reading file, Elapsed time - {watch.ElapsedMilliseconds}");

watch.Stop();

Console.ReadKey();

}

/// <summary>

/// Callback method

/// </summary>

/// <param name=" asyncResult "></param>

private static void EndRead(IAsyncResult asyncResult)

{

Console.WriteLine($" Managed Thread Id in endread is : {Thread.CurrentThread.ManagedThreadId}"); //// The managed thread identifier.

FileStream fs = (FileStream) asyncResult.AsyncState;

Int32 numBytesRead = fs.EndRead(asyncResult);

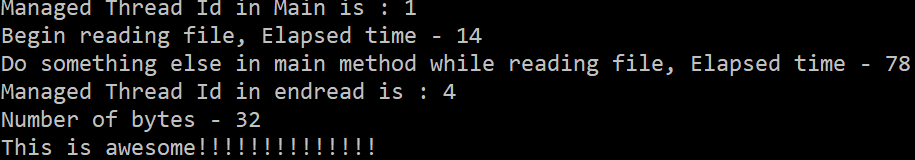
Console.WriteLine($" Number of bytes - {numBytesRead}");

Console.WriteLine(Encoding.UTF8.GetString(bytes));

fs.Close();

}

}



**Figure 6.14 – Output of file read using APM pattern with callback**

However, with the introduction of TAP, APM is no longer recommended, so it is good to know about APM but not recommended to use it.

**APM to TAP wrapper**

One use case where this can help is if there is a legacy/third party library which supports async operation using APM methods (BeginOperation/ EndOperation) , it is good to have understanding of APM so as to build a wrapper that can be used to expose APM operations as TAP operations.

This can be implemented by using Task.Factory.FromAsync method which takes Begin and End method as input and return a Task. For example, to read a file TAP wrapper method around it’s APM methods would look like below code

Task<int> ReadAsyncAPMWrapper(FileStream fs, byte[] buffer, int offset, int count)

{

return Task<int>.Factory.FromAsync(fs.BeginRead, fs.EndRead, buffer, offset, count, null);

}

This method can be called just like any other TAP method with await keyword. Something like

FileStream fs = new FileStream(@"../../../TextFile.txt", FileMode.Open, FileAccess.Read, FileShare.Read, bytes.Length, FileOptions.Asynchronous);

numBytesRead = await ReadAsyncAPMWrapper(fs, bytes, 0, bytes.Length, cts.Token);

However, there are limitations with this implementation in terms of things like cancellation token and better exception handling, logging. So, considering the same example above we will use BeginRead and EndRead of FileStream class and build a TAP wrapper on top of it by wrapping around APM methods in a Task and using TaskCompletionSource to signal completion or cancellation. The following code demonstrates how to implement a more sophisticated TAP wrapper over APM operations.

class Program

{

static Byte[] bytes = new Byte[100];

static CancellationTokenSource cts = new CancellationTokenSource();

static async Task Main(string[] args)

{

Console.WriteLine($"Managed Thread Id in Main is : {Thread.CurrentThread.ManagedThreadId}"); //// The managed thread identifier.

Stopwatch watch = new Stopwatch();

watch.Start();

FileStream fs = new FileStream(@"../../../TextFile.txt", FileMode.Open, FileAccess.Read, FileShare.Read, bytes.Length, FileOptions.Asynchronous);

Console.Write("Enter wait time in seconds before cancelling operation ");

int waitTime = Convert.ToInt32(Console.ReadLine());

cts.CancelAfter(waitTime \* 1000);

int numBytesRead = 0;

try

{

numBytesRead = await ReadAsyncAPMWrapper(fs, bytes, 0, bytes.Length, cts.Token);

Console.WriteLine("Operation completed");

}

catch (OperationCanceledException ex)

{

Console.WriteLine($"Operation cancelled - {ex.Message}");

}

finally

{

cts = null;

fs.Close();

Console.WriteLine($"Number of bytes - {numBytesRead}");

}

Console.ReadKey();

}

/// <summary>

/// TAP Wrapper over BeginRead and EndRead of FileStream

/// </summary>

/// <param name="fs"></param>

/// <param name="buffer"></param>

/// <param name="offset"></param>

/// <param name="count"></param>

/// <param name="token"></param>

/// <returns></returns>

static Task<int> ReadAsyncAPMWrapper(FileStream fs, byte[] buffer, int offset, int count, CancellationToken token)

{

var taskCompletionSource = new TaskCompletionSource<int>();

//Registering cancellation token, although this is not a elegant way to cancel as it doesn't handle IO resource cleanly.

// also this doesn't stop beginread

token.Register(() => taskCompletionSource.TrySetCanceled());

fs.BeginRead(buffer, offset, count, iAsyncResult =>

{

try

{

Thread.Sleep(5000); //If user input has wait more than this complete operation else cancel.

if (token.IsCancellationRequested)

{

throw new OperationCanceledException();

}

var state = iAsyncResult.AsyncState as FileStream;

var read = state.EndRead(iAsyncResult);

taskCompletionSource.TrySetResult(fs.EndRead(read));

}

catch (Exception exc)

{

taskCompletionSource.TrySetException(exc);

}

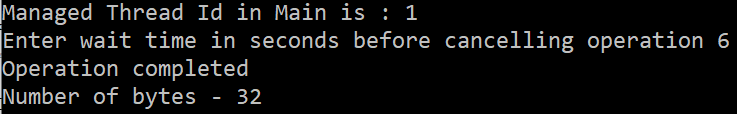
}, fs);

return taskCompletionSource.Task;

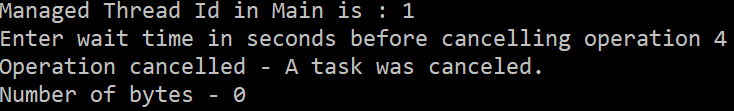
}

}

Here we are using TrySetResult and TrySetException methods to complete the Task<int> exposed through Task property of TaskCompletionSource class. As you can see rest of the bit in terms of exception handling and cancellation remains same like any other TAP method. Output of this code will look like below



**Figure 6.15 – Output TAP wrapper over APM methods**



**Figure 6.16 – Output TAP wrapper over APM methods, here job is cancelled before completion**

In Summary what we are doing here is taking APM methods, combining it to single method and wrapping them in a task.

Note - Although we are able to cancel this operation it’s not an elegant way to do due to limitations in APM methods, in reality we are not cancelling the file read operation but only cancelling wait on the operation.

**TAP to APM wrapper**

If we want to do vice versa i.e. converting TAP method to APM all we need to take task method and split it into 2 methods

* First one that can take AsyncCallback and State of the calling object and return IAsyncResult
* Second one accepting IAsyncResult from previous step

To build first method

* We will make use of TaskCompleteionSource class and it’s capability to create task instance
* Manually handle IsFaulted, IsHandled methods or successful result of TaskCompleteionSource task property to mirror results into task instance through TrySetException, TrySetCanceled and TrySetResult
* Pass TaskCompleteionSource task instance to callback provided (If any) and return same task instance.

So if we build a helper method on Task it would look like below

public static class TaskAPMExtension

{

/// <summary>

/// Generic extension method to convert TAP methods to APM

/// </summary>

/// <typeparam name="TResult"></typeparam>

/// <param name="task"></param>

/// <param name="asyncCallback"></param>

/// <param name="state"></param>

/// <returns></returns>

public static IAsyncResult TAPToApm<TResult>(this Task<TResult> task, AsyncCallback asyncCallback, object state)

{

var taskCompletionSource = new TaskCompletionSource<TResult>(state);

task.ContinueWith(delegate

{

if (task.IsFaulted)

{

taskCompletionSource.TrySetException(task.Exception.InnerExceptions);

}

else if (task.IsCanceled)

{

taskCompletionSource.TrySetCanceled();

}

else

{

taskCompletionSource.TrySetResult(task.Result);

}

if (asyncCallback != null)

{

asyncCallback(taskCompletionSource.Task);

}

}, CancellationToken.None, TaskContinuationOptions.None);

return taskCompletionSource.Task;

}

}

With the help of this helper method any TAP async method can be converted into begin operation of APM.

/// <summary>

/// Begin operation

/// </summary>

/// <param name="callback"></param>

/// <param name="state"></param>

/// <returns></returns>

static IAsyncResult BeginAPI(AsyncCallback callback, object state)

{

return GetStocksAsync(cts.Token).TAPToApm(callback, state);

}

For the second method, all we need is to take the response of first method, pass it as input parameter and get the result through Result property of Task<TResult>. So our implementation of calling method will look like below code along with cancellation and error handling

class Program

{

static CancellationTokenSource cts = new CancellationTokenSource();

static void Main(string[] args)

{

Stopwatch watch = new Stopwatch();

watch.Start();

IAsyncResult result = BeginAPI(null, null);

while (!result.IsCompleted) // Proceeding with doing some other operation while file is being read

{

if (watch.ElapsedMilliseconds % 3000 == 0)

{

cts.CancelAfter(15000);

Console.WriteLine($"Do something else in main method while receiving response from API, Elapsed time - {watch.ElapsedMilliseconds}");

}

}

string apiResponse = EndAPI(result);

Console.WriteLine($"API response - {apiResponse}, Elapsed time - {watch.ElapsedMilliseconds}");

//IAsyncResult result = BeginAPI(EndAPIUsingCallback, null); // Using callback

Console.ReadKey();

watch.Stop();

}

/// <summary>

/// Async method to retrieve data from API

/// </summary>

static async Task<string> GetStocksAsync(CancellationToken token)

{

using (HttpClient client = new HttpClient())

{

try

{

var response = await client.GetAsync("https://localhost:44394/api/Stocks", token);

response.EnsureSuccessStatusCode();

var content = await response.Content.ReadAsStringAsync();

Console.WriteLine("Data retrieved from API");

return content;

}

catch (Exception ex)

{

Console.WriteLine($"exception occured in API - {ex.Message}");

throw;

}

}

}

/// <summary>

/// Begin operation

/// </summary>

static IAsyncResult BeginAPI(AsyncCallback callback, object state)

{

return GetStocksAsync(cts.Token).TAPToApm(callback, state);

}

/// <summary>

/// End operation

/// </summary>

static string EndAPI(IAsyncResult asyncResult)

{

try

{

return ((Task<string>)asyncResult).Result;

}

catch (AggregateException ex)

{

List<Tuple<string, string>> errors = ex.Flatten().InnerExceptions.Select(x => new Tuple<string, string>(x.Message, x.StackTrace)).ToList();

int counter = 0;

foreach (Tuple<string, string> error in errors)

{

counter++;

Console.WriteLine($"{counter}).Error - {error.Item1} \n Innerstack \n {error.Item2} \n");

}

return $"Exception occured - {ex.Message}";

}

}

/// <summary>

/// Callback

/// </summary>

static void EndAPIUsingCallback(IAsyncResult asyncResult)

{

string apiResponse = ((Task<string>)asyncResult).Result;

Console.WriteLine($"API response - {apiResponse}");

}

}

As you can see I have implemented both variants one with callback and another without, Of Course there is more we could do here but our this is a good starting point if need arises to write an APM wrapper over TAP methods.

**Event-based Asynchronous pattern (EAP)**

Event based Asynchronous pattern is another way to add asynchronous capability to your methods, this pattern is introduced in .net 2.0 and the way we achieve asynchronous functionality is by splitting method into 2 parts

1. Creating a method typically suffixed with Asyn and is executed on a different thread.
2. An event confirming completion of async operation typically named with a suffix Completed

Additional methods that can be optionally added to async implementation is

* Cancellation support – To cancel async operation
* Progress support – To track progress of completion, typically named as Progresschanged event has an argument of type ProgressChangedEventArgs
* Returning incremental results – A capability to return results that are received say for example reading file line by line and then copying it to new file immediately instead of waiting for full file download
* IsBusy Property – If multiple invocations of method are not supported implement IsBusy property for caller to signal state of previous invocation, if our class supports multiple/parallel invocations of async operation IsBusy property shouldn’t be implemented

Let’s take an example of class that is used to calculate pi up to N number of places, the first thing that we need to define delegate for completion operation

public delegate void CalculatePiCompletedEventHandler(object sender, CalculatePiCompletedEventArgs e);

Implement class derived from AsyncCompletedEventArgs which will be input to the delegate, it will look something like below

public class CalculatePiCompletedEventArgs : AsyncCompletedEventArgs

{

//Any additional properties, they should be read only

public CalculatePiCompletedEventArgs(Exception e, bool canceled, object state) : base(e, canceled, state)

{

//set properties

}

}

Inside Calculatepi class add async operation completion event handler that will be triggered post completion of async operation

public event CalculatePiCompletedEventHandler CalculatepiCompleted;

At this point we can define our async method that accepts any specific input required for our operation and another parameter of type object which will be used to accept input user state, this parameter is used to identify completion during multiple invocation. This method along with synchronous method would look like below

public void CalculatePiAsync(int numSteps, object operationState)

{

AsyncOperation asyncOp = AsyncOperationManager.CreateOperation(operationState);

//code to handle multiple invocation

//Execute process Asynchronously

Task.Run(() => CalculatePi(numSteps, asyncOp));

}

void CalculatePi(int numsteps, AsyncOperation asyncOp)

{

//Code to calculate pi

CalculatePiCompletedEventArgs e = new CalculatePiCompletedEventArgs(null, false, asyncOp.UserSuppliedState);

CalculatePiCompletedEventArgs completion = e as CalculatePiCompletedEventArgs;

if (CalculatepiCompleted != null)

{

CalculatepiCompleted(this, completion); //raise completion event

}

}

However, this can be further optimized in terms of executing synchronous method through delegate and using a callback to raise completion event. So, we can define a delegate that matches signature of our synchronous operation in this case void CalculatePi(int numsteps, AsyncOperation asyncOp). So delegate will look like

private delegate void CalculationEventHandler(int numSteps, AsyncOperation asyncOp);

Secondly create an object of type that can raise completion event through a callback and then this callback needs to be executed on a thread from threadpool so that it can dispatch message to synchronization context. Object of class [SendOrPostCallback](https://docs.microsoft.com/en-us/dotnet/api/system.threading.sendorpostcallback) fulfills this requirement. The callback method needs to have a signature of accepting a single parameter of type object so that we can pass the state. Wiring of callback to [SendOrPostCallback](https://docs.microsoft.com/en-us/dotnet/api/system.threading.sendorpostcallback) delegate can be done through constructor. With this change the entire class will look like below

using System;

using System.Collections.Generic;

using System.Collections.Specialized;

using System.ComponentModel;

using System.Diagnostics;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace EAPCalculatepi

{

public delegate void CalculatepiCompletedEventHandler(object sender, CalculatepiCompletedEventArgs e);

public class Calculatepi

{

//Completion event handler

public event CalculatepiCompletedEventHandler CalculatepiCompleted;

//delegate to call the synchronous operation in async way (in this case we are using task.run)

private delegate void CalculationEventHandler(int numSteps, AsyncOperation asyncOp);

//Delegate to handle callback and raise completion event

private SendOrPostCallback onCompletedDelegate;

public Calculatepi()

{

onCompletedDelegate = new SendOrPostCallback(CalculationCompleted);

}

/// <summary>

/// callback method

/// </summary>

private void CalculationCompleted(object operationState)

{

CalculatepiCompletedEventArgs e = operationState as CalculatepiCompletedEventArgs;

if (CalculatepiCompleted != null)

{

CalculatepiCompleted(this, e);

}

}

//Dictionary to handle multiple tasks innovation, uniquely identify each operation as one dictionary element

private HybridDictionary parallelTasks = new HybridDictionary();

// Calculate pi async

void CalculatepiAsync(int numsteps, object operationState)

{

AsyncOperation asyncOp = AsyncOperationManager.CreateOperation(operationState);

//Locking for thread safety

lock (parallelTasks.SyncRoot)

{

if (parallelTasks.Contains(operationState))

{

throw new ArgumentException("User state parameter must be unique", "userState");

}

parallelTasks[operationState] = asyncOp;

}

CalculationEventHandler worker = new CalculationEventHandler(Calculatepivalue);

//Execute process Asynchronously

Task.Run(() => worker(numsteps, asyncOp));

}

// Synchronous method to calculate pi

void Calculatepivalue(int numsteps, AsyncOperation asyncOp)

{

Stopwatch timer = new Stopwatch();

timer.Start();

numsteps++;

uint[] value = new uint[numsteps \* 10 / 3 + 2];

uint[] rem = new uint[numsteps \* 10 / 3 + 2];

uint[] pi = new uint[numsteps];

for (int j = 0; j < value.Length; j++)

value[j] = 20;

for (int i = 0; i < numsteps; i++)

{

uint carryForward = 0;

for (int j = 0; j < value.Length; j++)

{

uint number = (uint)(value.Length - j - 1);

uint pow = number \* 2 + 1;

value[j] += carryForward;

uint quotient = value[j] / pow;

rem[j] = value[j] % pow;

carryForward = quotient \* number;

}

pi[i] = (value[value.Length - 1] / 10);

rem[value.Length - 1] = value[value.Length - 1] % 10; ;

for (int j = 0; j < value.Length; j++)

value[j] = rem[j] \* 10;

}

var result = "";

uint c = 0;

for (int i = pi.Length - 1; i >= 0; i--)

{

pi[i] += c;

c = pi[i] / 10;

result = (pi[i] % 10).ToString() + result;

Thread.Sleep(10);

}

result = result.Substring(0, 1) + "." + result.Substring(1, result.Length - 1);

lock (parallelTasks.SyncRoot)

{

parallelTasks.Remove(asyncOp.UserSuppliedState);

}

//raise callback

CalculatepiCompletedEventArgs e = new CalculatepiCompletedEventArgs(result, timer.ElapsedMilliseconds, null, false, asyncOp.UserSuppliedState);

asyncOp.PostOperationCompleted(onCompletedDelegate, e);

timer.Stop();

}

}

public class CalculatepiCompletedEventArgs : AsyncCompletedEventArgs

{

public string Result { get; private set; }

public long TimeTaken { get; private set; }

public CalculatepiCompletedEventArgs(string value, long TimeTaken, Exception e, bool canceled, object state) : base(e, canceled, state)

{

this.Result = value;

this.TimeTaken = TimeTaken;

}

}

}

Consuming it in a console application

class Program

{

static void Main(string[] args)

{

Calculatepi pi = new Calculatepi();

pi.CalculatepiCompleted += new CalculatepiCompletedEventHandler(Pi\_CalculatePrimeCompleted);

Console.WriteLine("Calculating pi to 1000 places started");

pi.CalculatepiAsync(1000, 1000);

Console.WriteLine("Calculating pi to 900 places started");

pi.CalculatepiAsync(900, 900);

Console.WriteLine("do something else");

Console.ReadKey();

}

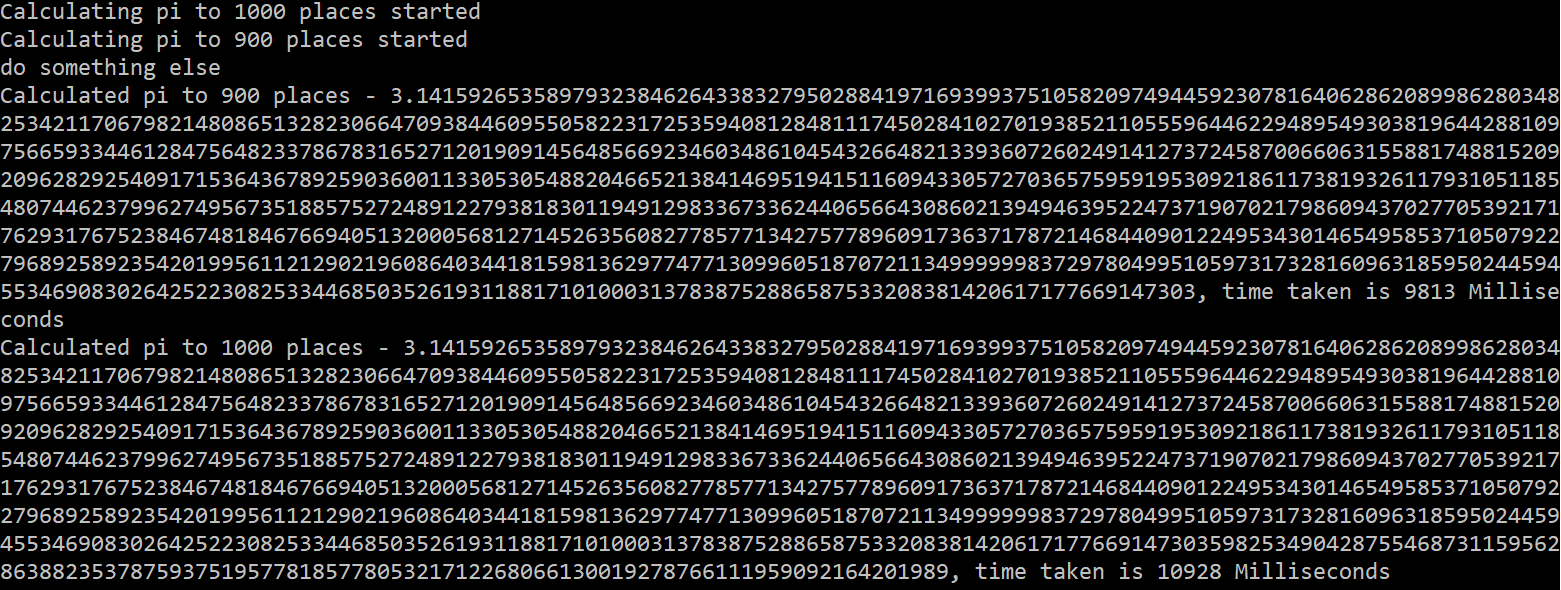
static void Pi\_CalculatePrimeCompleted(object sender, CalculatepiCompletedEventArgs e)

{

Console.WriteLine($"Calculated pi to {e.UserState.ToString()} places - {e.Result}, time taken is {e.TimeTaken.ToString()} Milliseconds");

}

}



**Figure 6.17 – Output calculate pi using EAP**

**EAP to TAP wrapper**

Like TAP wrapper over APM wrapper asynchronous operation implemented using EAP can be implemented using TaskCompletionSource class. SetException, SetCanceled, SetResult methods of TaskCompletionSource class needs to be handled based on exception, cancellation or successful completion of operation, finally Task property of TaskCompletionSource class is used to return from the wrapper method.

Taking the above example first let’s modify Calculatepi to implement cancellation

* Added a cancellation method to Calculatepi class where we need remove cancelled task from our dictionary.
* Call this method to break pi calculation (called it in one of the for loop to stop calculating pi) and raise completion event and also pass cancelled status flag to CalculatepiCompletedEventArgs.

// This method cancels a pending asynchronous operation.

public void CancelAsync(object operationState)

{

AsyncOperation asyncOp = parallelTasks[operationState] as AsyncOperation;

if (asyncOp != null)

{

lock (parallelTasks.SyncRoot)

{

parallelTasks.Remove(operationState);

}

}

}

//Utility method to check the task status

private bool TaskCanceled(object operationState)

{

return (parallelTasks[operationState] == null);

}

Next thing we need to implement is the wrapper method using CalculatepiAsync and CalculationEventHandler of our Calculatepi class.It will look something like below

private Task<CalculatepiCompletedEventArgs> TAPWrappertoAPMAsync(Calculatepi calculatepi, int numsteps, object sender, object operationState, CancellationToken token)

{

var tcs = new TaskCompletionSource<CalculatepiCompletedEventArgs>();

//Delegate on cancellation token when there is a cancellation, executes on calling thread's context in this case UI

token.Register(() =>

{

calculatepi.CancelAsync(operationState);

});

calculatepi.CalculatepiCompleted += (\_, e) =>

{

if (e.Cancelled)

tcs.TrySetCanceled();

else if (e.Error != null)

tcs.TrySetException(e.Error);

else

tcs.TrySetResult(e);

};

// Register for the event and start the operation.

calculatepi.CalculatepiAsync(numsteps, sender, operationState);

return tcs.Task;

}

Code is pretty much self explanatory where we are making use of TaskCompletionSource and a handler of type CalculatepiCompletedEventHandler to set appropriate status of TaskCompletionSource and then finally return Task. As you can see this can be further extended to an extension or a generic method.

Calling this on a button click of a WPF application

public partial class MainWindow : Window

{

CalculatepiCompletedEventArgs calculatepiCompletedEventArgs = null;

CancellationTokenSource cts = null;

private async void Calculatepi1000TAP\_Click(object sender, RoutedEventArgs e)

{

Calculatepi calculatepi = new Calculatepi();

calculatepi1000TAP.Content = "Cancel";

//On clicking of Cancel checking to cancel operation

if (cts != null)

{

cts.Cancel();

cts = null;

return;

}

cts = new CancellationTokenSource();

//Cancellation needs to be handled gracefully

try

{

calculatepiCompletedEventArgs = await TAPWrappertoAPMAsync(calculatepi, 1000, sender, "Calculate pi (1000) TAP", cts.Token);

output.Text += $"{calculatepiCompletedEventArgs.UserState.ToString()} - {calculatepiCompletedEventArgs.Result}, time taken is {calculatepiCompletedEventArgs.TimeTaken.ToString()} Milliseconds{Environment.NewLine}";

}

catch (OperationCanceledException)

{

output.Text += $"Calculate pi(1000) TAP is cancelled";

}

finally

{

cts = null;

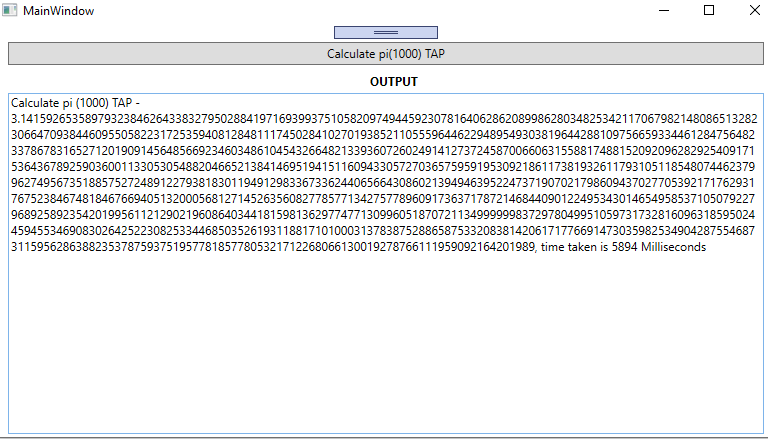
calculatepiCompletedEventArgs = null;

}

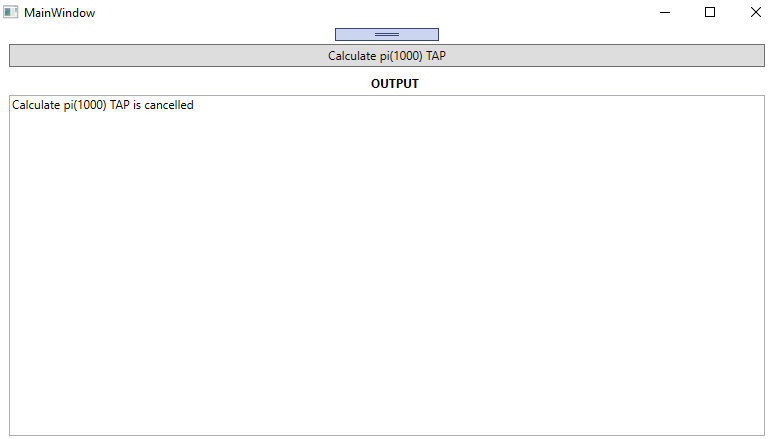
calculatepi1000TAP.Content = "Calculate pi(1000) TAP";

}

}



**Figure 6.18 – Output calculate pi using TAP wrapper over EAP**



**Figure 6.19 – Output cancelling pi calculation using TAP wrapper over EAP**

**Summary**

In this chapter we have seen various patterns that can be used to implement asynchronous operations, we have seen

* Task-based Asynchronous pattern(TAP) in detail
  + Exception handling
  + Cancellation
  + Report progress

And legacy patterns

* Event-based Asynchronous pattern (EAP)
* Asynchronous programming model(APM).

As mentioned, if it’s a new implementation we should go with TAP, however it is good to know EAP and APM as it would help to implement TAP wrappers on these patterns. With the understanding of these patterns you can identify areas in your application that can be parallelized efficiently.

**Questions:**

1. Using Progress in TAP create a sample to pause and resume file/data download
2. See if you can close stream in case of cancellation while implementing APM to TAP wrapper
3. Convert console application of calculating API into a WPF app and implement progress.