**Chapter 6: The Threading Patterns**

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

**Overview**

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

**A button click event on a win form loading data from API synchronously**

private async 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";

}

**A button click event on a win form loading data from API asynchronously**

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 Figure 6.2 await keyword helps is getting the result from asynchronous operation once data is available without blocking 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.

**CPU Bound vs I/O Bound**

When implementing asynchronous code specially 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 source 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

using System;

using System.Net.Http;

using System.Threading;

using System.Threading.Tasks;

namespace CPUBoundvsIOBound

{

class Program

{

static async Task Main()

{

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

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

AvailableThreads();

await Task.Run(() => GetStocks());

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

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

AvailableThreads();

await Task.Run(() => DoExpensiveCalculation());

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

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

AvailableThreads();

Console.ReadLine();

}

/// <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);

}

/// <summary>

/// Async method to retrieve data from API

/// </summary>

static async Task GetStocks()

{

using (HttpClient client = new HttpClient())

{

try

{

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

response.EnsureSuccessStatusCode();

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

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

}

catch (Exception ex)

{

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

}

}

}

/// <summary>

/// Method performing high CPU intense calculation

/// </summary>

static async Task<double> DoExpensiveCalculation()

{

Console.WriteLine("Start CPU Bound asynchronous task");

float calculation = 0;

var output = await Task.Run(() =>

{

for (int i = 0; i < 100; i++)

{

calculation = calculation \* 20;

}

return calculation;

});

Console.WriteLine("Finished CPU bound Task");

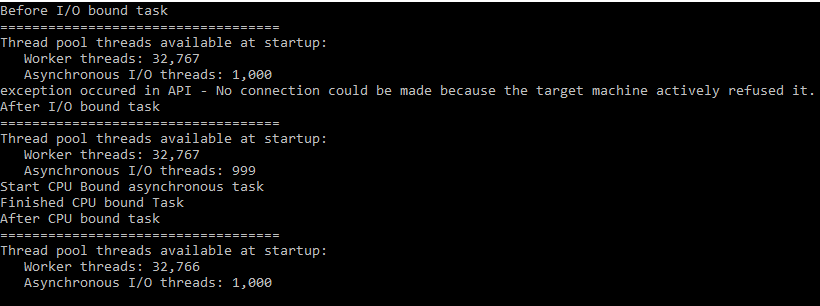
return output;

}

}

}

Running the code above will give following output, here you can see that although we are calling a background thread for CPU intense operation it has used a worker thread which is ok for client side application for things like unblocking UI, However for a ASP.Net application this is nothing different than running operation synchronously as there is no added benefit of assigning a dedicated thread for CPU bound operation.



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

Taking 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.

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.

**Exception Handling**

**Basic 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 oversimplification 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.

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;

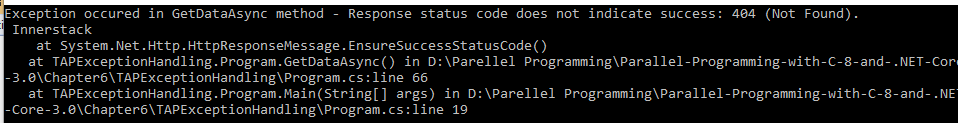
}

}

}

}

}



**Fig 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, adding below code to the catch block of main method will give same output

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}");

}

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, handling that wouldn’t be any different i.e. use await and original exception is unwrapped and is propagated to caller. This is illustrated in below example

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 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();

}

/// <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;

}

}

}

/// <summary>

/// Dummy nested method

/// </summary>

/// <returns></returns>

static async Task<String> GetDataAsyncNested()

{

return await GetDataAsync();

}

/// <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

{

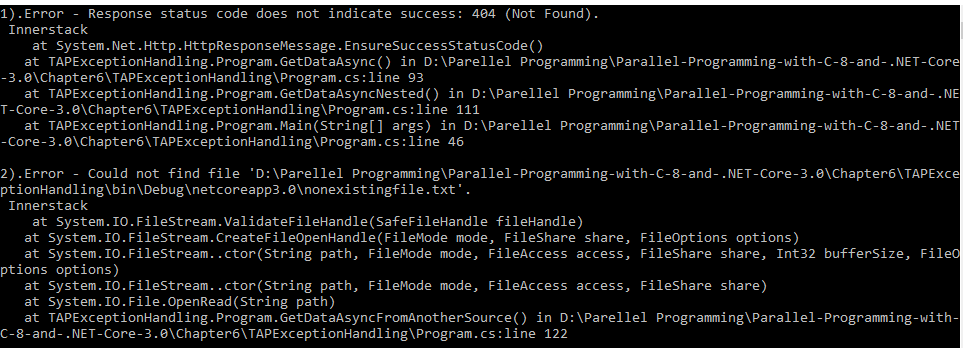
throw;

}

}

}

}



**Fig 6.3 – Nested exception handling**

**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. Following code shows an example used for async operations that are implemented without await and how AggregateException can be used to loop through multiple exceptions returned from various async operations.

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)

{

var tasks = new List<Task>();

var task = GetDataAsync();

tasks.Add(task);

task = Task.Run(() => DoHighCPUIntense());

tasks.Add(task);

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();

}

/// <summary>

/// Async method doing high CPU operation

/// </summary>

/// <returns></returns>

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();

});

output.Wait();

return string.Empty;

}

/// <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;

}

}

}

}

}

Output of this code is exactly same as the one in Fig 6.3.

**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 is needed 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

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;

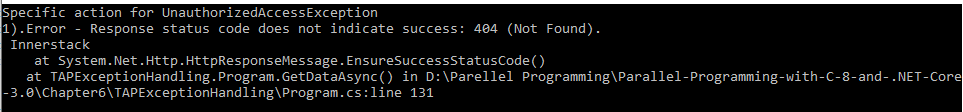
});

}

return string.Empty;

}

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

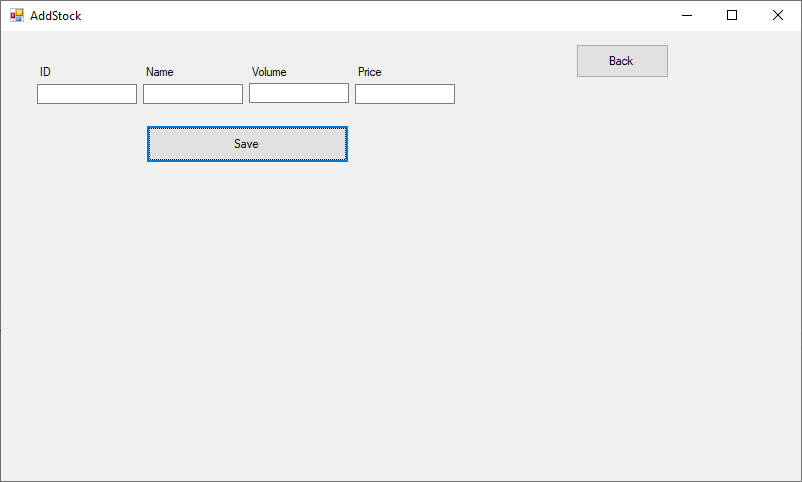


**Fig 6.4 – Using handle method**

**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 returing 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 see a simple winform to see this behavior where we have windows form as shwn in Fig 6.5



**Fig 6.4 – Windows form to save data through an API**

On click of save we need to make a fire and forget API call to save data.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 occured in SaveDataAsyncVoid method - {ex.Message} \n Innerstack \n {ex.StackTrace}";

}

finally

{

setID.Text = "";

setStockName.Text = "";

setStockPrice.Text = "";

setStockVolume.Text = "";

}

}

/// <summary>

/// Async method to save data through API

/// </summary>

private async void SaveDataAsyncVoid()

{

using (HttpClient client = new HttpClient())

{

Stock data = new Stock()

{

Id = Convert.ToInt32(setID.Text),

StockName = setStockName.Text,

Price = Convert.ToDouble(setStockPrice.Text),

TradeDate = DateTime.Today.Date.AddDays(-1),

Volume = Convert.ToInt32(setStockVolume.Text)

};

var myContent = JsonConvert.SerializeObject(data);

var buffer = System.Text.Encoding.UTF8.GetBytes(myContent);

var byteContent = new ByteArrayContent(buffer);

byteContent.Headers.ContentType = new MediaTypeHeaderValue("application/json");

var response = await client.PostAsync("https://localhost:44394/api/stocks", byteContent);

if (response.StatusCode == HttpStatusCode.InternalServerError)

{

string error = await response.Content.ReadAsStringAsync();

throw new Exception(error);

}

errorMessage.BeginInvoke((MethodInvoker)delegate () {

errorMessage.Text = "Data saved successfully";

});

}

}

Once you click save if there is an exception from API you can see that app crashes and that’s because an exeption is raised on the SynchnronizationContext of the UI thread. So to handle this either we need build custom SynchnronizationContext 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 explictly 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 advantage with TAP is the ease with which asynchronous operation can be cancelled. Cancelling an operation plays 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 calling method. Cancellation operation throws 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 applciation 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/Canacel checking to cancel opearation 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 retieve 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();

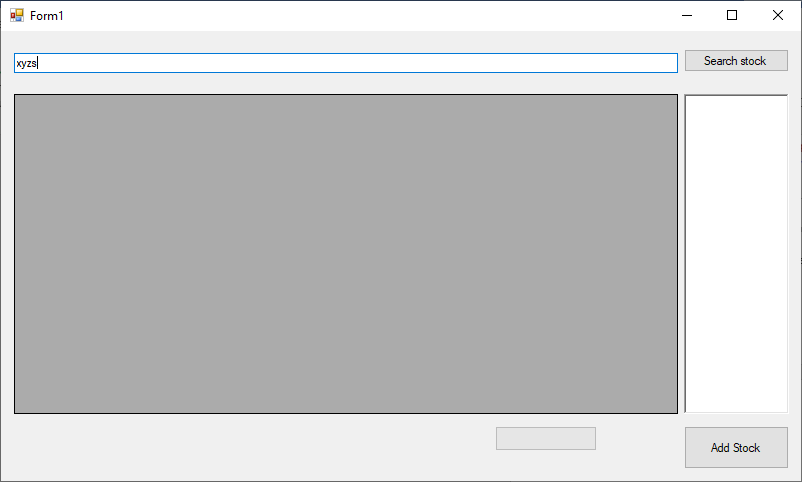
var data = JsonConvert.DeserializeObject<IEnumerable<Stock>>(content);

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

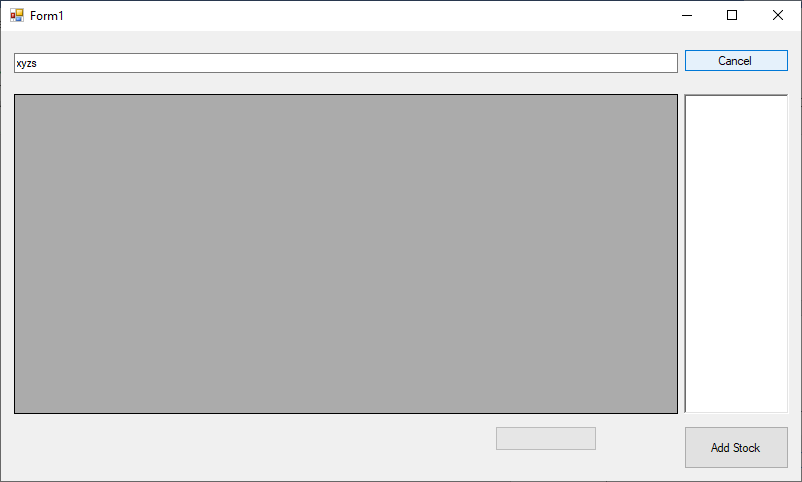
}

return bindingSource1;

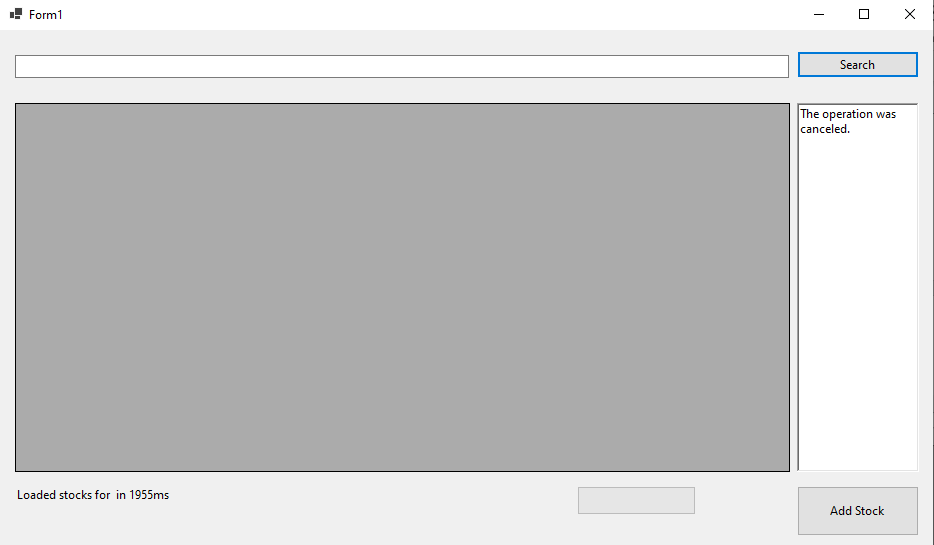
}



**Fig 6.5 – Windows form to search stock – “xyzs”**



**Fig 6.5 – Windows form showing cancel button**



**Fig 6.6 – Windows form after cancelling search operation**

In 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 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 cire 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.

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

However there could be cases where there is need that async operation doesn’t through exception (OperationCanceledException) but return normally. However, in this case calling method need not handle such exception For example Calculating prime numbers lesser 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 it’s output is shown in Fig 6.7

/// <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;

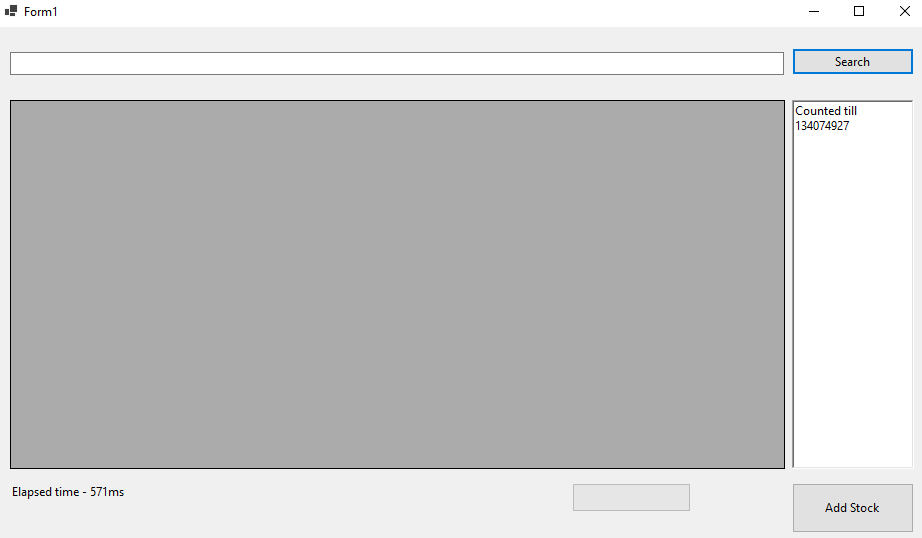
#endregion

# region "Scenario 2"

highCPUCount = await DoHighCPUIntense(cts.Token);

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

#endregion



**Fig 6.7 – 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 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**

**Paralleling with TAP (WhenAll/WhenAny/ContinueWith)**

**Throttling**

**Retry**

**The awaitable and awaiter GetAwaiter()**

[**https://devblogs.microsoft.com/pfxteam/asyncawait-faq/**](https://devblogs.microsoft.com/pfxteam/asyncawait-faq/)

[**https://devblogs.microsoft.com/pfxteam/await-anything/**](https://devblogs.microsoft.com/pfxteam/await-anything/)

**Interop with APM, TAP to APM**