C# tasks

C# task is one of the central elements of the task-based asynchronous pattern first introduced in the .NET Framework 4.

C# task object typically executes asynchronously on a thread pool thread rather than synchronously on the main application thread.

Although, we can directly offload the work to thread pool using the queue user work item method. However, this method has its weaknesses as we can't tell whether the operation has finished or what a return value is.

This is where a C# task can be helpful. The C# task can tell you if the work is completed and if the operation returns a result. A Task is an object that represents some work that should be done.

C# tasks provide a sophisticated way to handle async or parallel operation by providing various options like,

- Ability to cancel an ongoing operation
- Return resulting value from operation (like a method functions)
- Easy Exception Handling
- High-Level constructs like a parallel loop
- task continuation

Using C# Tasks

To get started with **Tasks in C#** you have to create an object of Task Class, which is available in namespace `System.Threading.Tasks` and provide the code to be executed within the task as the task action parameter.

C# Task Example

Let's see an example,

```
1 using System;

2 using System.Collections.Generic;
```

```
 3 using System.Linq;
 4 using System.Text;
 5 using System.Threading;
 6 using System.Threading.Tasks;
 7
 8 namespace _01_Getting_Started_With_Task
 9 {
10     class Program
11     {
12         static void Main(string[] args)
13         {
14             //create task object and pass anonymous method
15             //to task constructor parameter as work to do
16             //within the task
17             Task t = new Task(() =>
18               {
19                   for (int i = 0; i < 100; i++)
20                   {
21                       //print task t thread id
22                       var threadId =
Thread.CurrentThread.ManagedThreadId;
23                       Console.WriteLine("Task Loop Current Thread Id:" +
threadId);
```

```
24                      }
25                  });
26
27              //start task t execution
28              t.Start();
29
30              for (int i = 0; i < 100; i++)
31              {
32                  //print main thread id
33                  var threadId = Thread.CurrentThread.ManagedThreadId;
34                  Console.WriteLine("Main Loop Current Thread Id " +
threadId);
35              }
36
37              //wait for task t to complete its execution
38              t.Wait();
39
40              Console.WriteLine("Press enter terminate the process!");
41              Console.ReadLine();
42          }
43      }
44  }
```

Let’s see what’s happening in the above program line by line.

Here, we are creating the Task `t` along with the code to execute within the task. However, Task `t` starts its execution after calling `t.Start()` method.

So,

After calling `t.Start()`, our program is split into two programs that are executing in the parallel codestream also known as a fork or split in the program.

So basically code in task and code below the task’s start method, both are executing together. You can’t see this but this is happening and you have to imagine this, that now your program is executing two parallel codestreams.

Also, you must have noticed that we passed an anonymous method to the Task constructor parameter, as a code to be executed within the task. This we can do because Task Class constructor takes `Action` as a parameter which is nothing but a delegate with the void return type.

Execution Model Of A Task

Let’s discuss the basic execution model of a task,

In the example above, we created a task and provide a basic operation to be performed by the task.

A task scheduler is responsible for starting the Task and managing it. By default, the Task scheduler uses threads from the thread pool to execute the Task.

Tasks can be used to make your application more responsive. If the thread that manages the user interface offloads work to another thread from the thread pool, it can keep processing user events and ensure that the application can still be used.

But it doesn’t help with scalability. If a thread receives a web request and it would start a new Task, it would just consume another thread from the thread pool while the original thread waits for results.

Executing a Task on another thread makes sense only if you want to keep the user interface thread free for other work or if you want to parallelize your work on to multiple processors.

Also, In the above example, you may have seen the below code,

```
1 //wait for task t to complete its execution

2 t.Wait();
```

Calling `Wait` method of the task is equivalent to calling the `Join` method on a thread. When the `Wait` method is called within the `Main` method the main thread pauses its execution until the task `t` completes its execution.

C# Task That Returns A Value

The .NET Framework also has the generic version of task class `Task<T>` that you can use if a Task should return a value. Here `T` is the data type you want to return as a result. The below code shows how this works.

```
1  using System;
2  using System.Threading.Tasks;
3
4  namespace Example2
5  {
6      class Program
7      {
8          static void Main(string[] args)
9          {
10             Task<int> t = Task.Run(() =>
11             {
12                 return 32;
```

```
13              });
14              Console.WriteLine(t.Result); // Displays 32
15
16              Console.WriteLine();
17              Console.WriteLine("Press Enter to terminate!");
18              Console.ReadLine();
19          }
20      }
21  }
```

Attempting to read the `Result` property on a Task will force the thread that's trying to read the result to wait until the task is finished, which is equivalent to calling `Join` method on a thread and calling `Wait` method in the task as mentioned before.

As long as the Task has not finished, it is impossible to give the result. If the task is not finished, this call will block the current thread.

How To Avoid Blocking Of Current Thread? How To Be Responsive?

As I mentioned at the start of this chapter,

Tasks can be used to make your application more responsive.

But, until now the only thing I told you is this,

Calling `Wait` method of the task is equivalent to calling the `Join` method on a thread. When the `Wait` method is called within the `Main` method the main thread pauses its execution until the task `t` completes its execution.

and this,

Attempting to read the `Result` property on a Task will force the thread that's trying to read the result to wait until the task is finished.

Then how in the world task can help be responsive? Your question is right and below is the answer that you are looking for.

Adding A Continuation

Another great feature that task supports is the continuation. This means that you can execute another task as soon as the first task finishes. Thus, you can avoid the block that we discussed before. This method is similar to calling the callback method when a certain operation is finished.

Below is an example of creating such a continuation.

```
using System;

 2 using System.Threading.Tasks;

 3

 4 namespace Example3

 5 {

 6     class Program

 7     {

 8         static void Main(string[] args)

 9         {

10             Task<int> t = Task.Run(() =>

11             {

12                 return 32;

13             }).ContinueWith((i) =>

14             {
```

```
15                  return i.Result * 2;
16              });
17
18              t.ContinueWith((i) =>
19              {
20                   Console.WriteLine(i.Result);
21              });
22
23              Console.WriteLine("Press Enter to terminate!");
24              Console.ReadLine();
25          }
26      }
27  }
```

Scheduling Different Continuation Tasks

The ContinueWith method has a couple of overloads that you can use to configure when the continuation will run. This way you can add different continuation methods that will run when an exception happens, the Task is cancelled, or the Task completes successfully. The below code shows how to do this.

```
1  using System;
2  using System.Threading.Tasks;
3
4  namespace Example_4
```

```
5 {
6     class Program
7     {
8         static void Main(string[] args)
9         {
10            Task<int> t = Task.Run(() =>
11            {
12                return 32;
13            });
14            t.ContinueWith((i) =>
15            {
16                Console.WriteLine("Canceled");
17            }, TaskContinuationOptions.OnlyOnCanceled);
18            t.ContinueWith((i) =>
19            {
20                Console.WriteLine("Faulted");
21            }, TaskContinuationOptions.OnlyOnFaulted);
22            var completedTask = t.ContinueWith((i) =>
23            {
24                Console.WriteLine(i.Result);
25                Console.WriteLine("Completed");
```

```
26              }, TaskContinuationOptions.OnlyOnRanToCompletion);
27
28              Console.WriteLine("Press Enter to terminate!");
29              Console.ReadLine();
30          }
31      }
32  }
```

Using task with continuation is a great way to create a responsive application that doesn't block the main thread. In desktop applications like Windows Forms and WPF, we can use this feature of the task to create a very responsive application that doesn't block the UI thread.

Demo Of A Non-Responsive Application

In this application we simply calculate the nth term of the Fibonacci series. In the application when we put some higher value (ex: "900000000") for calculating nth term we see that application UI get lock up until the calculation is going on and after a while when execution is complete we see the result along with the time required by the program to get the result.

Here is the code of the application,

```
1  using System;
2  using System.Diagnostics;
3  using System.Windows.Forms;
4
5  namespace _02_Program_For_Responsiveness
6  {
```

```
 7      public partial class Form1 : Form
 8      {
 9          public Form1()
10          {
11              InitializeComponent();
12          }
13
14          private void Calculate_Click(object sender, EventArgs e)
15          {
16              var stopWatch = new Stopwatch();
17              stopWatch.Start();
18              textBox2.Text = Fibo(textBox1.Text).ToString();
19              stopWatch.Stop();
20              label2.Text = (stopWatch.ElapsedMilliseconds /
1000).ToString();
21          }
22
23          private ulong Fibo(string nthValue)
24          {
25              try
26              {
27                  ulong x = 0, y = 1, z = 0, nth, i;
```

```
28                  nth = Convert.ToUInt64(nthValue);
29                  for (i = 1; i <= nth; i++)
30                  {
31                      z = x + y;
32                      x = y;
33                      y = z;
34                  }
35
36                  return z;
37              }
38              catch { }
39
40              return 0;
41          }
42      }
43  }
```

Adding Tasks For Responsiveness

Let's add a task in our `Calculate_Click` method to make it responsive,

```
1  private void Calculate_Click(object sender, EventArgs e)
2          {
3              var task=new Task(() =>
```

```
 4              {
 5                  var stopWatch = new Stopwatch();
 6                  stopWatch.Start();
 7                  textBox2.Text = Fibo(textBox1.Text).ToString();
 8                  stopWatch.Stop();
 9                  label2.Text = (stopWatch.ElapsedMilliseconds /
1000).ToString();
10              });
11
12              task.Start();
13          }
```

When you run this version of the application you will find that it's not updating values; Also if you look into the debug window of visual studio you will notice that there is InvalidOperationExecption raised each time you hit the calculate button.

Why?

WPF and most Windows UI frameworks have something called "Thread affinity". This means you can't change UI stuff on any thread that isn't the main UI thread.

Let's try to solve this problem,

To solve this problem one thing we can do is we can specify the .NET to run this particular task in UI thread which is our current thread instead of running into a separate thread by using the task scheduler parameter in `task.Start()` method like this,

```
1 task.Start(TaskScheduler.FromCurrentSynchronizationContext());
```

Here is the complete `Calculate_Click` method example,

```
 1  private void Calculate_Click(object sender, EventArgs e)

 2          {

 3              var task=new Task(() =>

 4              {

 5                  var stopWatch = new Stopwatch();

 6                  stopWatch.Start();

 7                  textBox2.Text = Fibo(textBox1.Text).ToString();

 8                  stopWatch.Stop();

 9                  label2.Text = (stopWatch.ElapsedMilliseconds /
1000).ToString();

10              });

11

12
task.Start(TaskScheduler.FromCurrentSynchronizationContext());

13          }
```

However, when you run this version of the application you will find that the user interface is once again lock up.

Again Why?

Because, we're running the task on the same UI thread, consuming the same UI thread's execution cycle until the task execution is done, which is essentially the entire time which locks up the user interface and thus we're back to where we started.

Correct solution:

The key solution to above problem is that we want to do the CPU intense work in one task which will run on worker thread by default and run the UI related work on another task, and the key is here that we want to run the second task only after when the first task is done with its execution. To achieve this functionality task provides the `ContinueWith` method which does the same thing here is the solution.

Also,

You have to use a `TaskScheduler` associated with the current UI SynchronizationContex as the second parameter in `task.ContinueWith` to run a new continuation task on the UI thread.

Let's see how the final code will look like,

```
1 private void Calculate_Click(object sender, EventArgs e)

2         {

3             Stopwatch stopWatch = new Stopwatch();

4             string result="";

5

6             var task=new Task(() =>

7             {

8                 stopWatch.Start();

9                 result = Fibo(textBox1.Text).ToString();

10            });

11

12            task.ContinueWith((previousTask) =>

13            {
```

```
14                  textBox2.Text = result;

15                  stopWatch.Stop();

16                  label2.Text = (stopWatch.ElapsedMilliseconds /
1000).ToString();

17                  stopWatch.Reset();

18              },

19              TaskScheduler.FromCurrentSynchronizationContext()

20              );

21

22              task.Start();

23          }
```

And that’s how it’s done!

However, there are still some other improvements that can be done in the above code.

Async And Await In C#

C# 5.0 introduced the async and await keywords. These keywords let you write asynchronous code that has the same structure and simplicity as synchronous code, as well as eliminating the “plumbing” of asynchronous programming.

Awaiting

The await keyword simplifies the attaching of C# Task continuations.

Starting with a basic scenario, the compiler expands:

```
1 var result = await expression;

2 statement(s);
```

into something functionally similar to:

```
1 var awaiter = expression.GetAwaiter();

2 awaiter.OnCompleted (() =>

3 {

4     var result = awaiter.GetResult();

5     statement(s);

6 });
```

The compiler also emits code to short-circuit the continuation in case of synchronous completion.

Awaiting in a UI

To demonstrate the above idea, I made a small WPF application that fetches data from the URL given in the textbox. You can find the complete project here.

```
1 using System;
2 using System.Net.Http;
3 using System.Threading.Tasks;
4 using System.Windows;
5
6 namespace WpfApp
7 {
8     public partial class MainWindow : Window
9     {
10         public MainWindow()
11         {
12             InitializeComponent();
13         }
14
15         private void Button_Click(object sender, RoutedEventArgs e)
```

```
16      {
17          var url = textBox.Text;
18          var isValidUrl =  Uri.TryCreate(url, UriKind.Absolute, out _);
19          if (!isValidUrl)
20          {
21              textBlock.Text = "Given url is not valid.";
22              return;
23          }
24
25          var currentContext = TaskScheduler.FromCurrentSynchronizationContext();
26          var httpClient = new HttpClient();
27
28          var responseTask = httpClient.GetAsync(url);
29          //First continuation start
30          responseTask.ContinueWith(r => {
31              try
32              {
33                  var response = r.Result;
34                  response.EnsureSuccessStatusCode();
35
36                  var dataTask = response.Content.ReadAsStringAsync();
```

```
37                //Second continuation start
38                dataTask.ContinueWith(d => {
39                    textBlock.Text = d.Result;
40                }, currentContext);
41                //Second continuation end
42            }
43            catch (Exception ex)
44            {
45                textBlock.Text = ex.Message;
46            }
47        });
48        //First continuation ends
49    }
50 }
51 }
```

Here `responseTask.ContinueWith` what simply does is that it callback the action, defined as the lambda expression, after the `responseTask` operation is finished.

Callbacks are not all bad; they worked — they still do. But, what happens if we have a callback inside a callback, inside a callback — you get the point. It gets messy and unmaintainable really quick.

Async Await Keyword

A very common thing to first try out when you encounter the asynchronous method in .NET is to simply mark your parent method with the async keyword. Let's go ahead and try that.

Also, let’s remove all the continuation from the code and directly use the .Result property to access the result from our asynchronous task operation.

Let’s see how that affects our application.

```
using System;
using System.Net.Http;
using System.Windows;

namespace WpfApp
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void Button_Click(object sender, RoutedEventArgs e)
        {
            var url = textBox.Text;
            var isValidUrl =  Uri.TryCreate(url, UriKind.Absolute, out _);
            if (!isValidUrl)
            {
```

```
20              textBlock.Text = "Given url is not valid.";
21              return;
22          }
23
24          var httpClient = new HttpClient();
25          var response = httpClient.GetAsync(url).Result;
26          try
27          {
28
29              response.EnsureSuccessStatusCode();
30              var data = response.Content.ReadAsStringAsync().Result;
31              textBlock.Text = data;
32          }
33          catch (Exception ex)
34          {
35              textBlock.Text = ex.Message;
36          }
37      }
38  }
39 }
```

You'll notice here that I can mark my Button_Click handler as async. And if you run the application and see if this affected the performance of our application, you'll quickly notice

that the application UI gets locks up. Let's jump into the code and discuss about why this is still not an asynchronous operation.

Visual Studio will tell us that this method is marked as async, but it lacks the await keyword. So the code inside this method will still run synchronously, and that's a big problem because we want to leverage the asynchronous principles.

Why is this a problem?

Now, in the above code you may have seen the following line:

```
1 var response = httpClient.GetAsync(url).Result;
```

Here GetAsync returns a task of an HttpResponseMessage, a task is a representation of our asynchronous operation. This asynchronous operation happens on a different thread.

So if we call .Result, which is one of the first things that people try to get the Result out of their asynchronous operation, this is actually a really bad idea.

It will actually block the thread until this Result is available, so this is problematic because this means that code will run synchronously.

Actually, what we need to do is to make sure that whenever we encounter the async keyword, we also have the await keyword inside that same method. Like this,

```
1 var response = await httpClient.GetAsync(url);
```

The await keyword is a way for us to indicate that we want to get the Result out of this asynchronous operation only once the data is available without blocking the current thread. So the above code gives us the HttpResponseMessage.

Also,

While reading the content from the response you'll find that ReadAsString is also an asynchronous operation, and it also hints us here that we need to await this as well.

```
1 var data = await response.Content.ReadAsStringAsync();
```

We could, of course, say ReadAsStringAsync and then call the Result property, but this would block again and make this code run synchronously, and in a lot of cases, calling `.Result` or `.Wait` will, in fact, deadlock the application. So avoid calling `.Result` or `.Wait`.

Let's see the final result with all the changes we did,

```
1 using System;
2 using System.Net.Http;
3 using System.Windows;
4
5 namespace WpfApp
6 {
7     public partial class MainWindow : Window
8     {
9         public MainWindow()
10        {
11            InitializeComponent();
12        }
13
14        private async void Button_Click(object sender, RoutedEventArgs e)
15        {
16            var url = textBox.Text;
17            var isValidUrl =  Uri.TryCreate(url, UriKind.Absolute, out _);
18            if (!isValidUrl)
```

```
19          {
20              textBlock.Text = "Given url is not valid.";
21              return;
22          }
23
24          var httpClient = new HttpClient();
25          var response = await httpClient.GetAsync(url);
26          try
27          {
28
29              response.EnsureSuccessStatusCode();
30              var data = await response.Content.ReadAsStringAsync();
31              textBlock.Text = data;
32          }
33          catch (Exception ex)
34          {
35              textBlock.Text = ex.Message;
36          }
37      }
38  }
39 }
```

So,

The await keyword, allows us to retrieve the result out of our asynchronous operation when that's available. It also makes sure that there are no exceptions or problems with the task that it's currently awaiting. So not only is the await keyword a great way for us to get the Result out of the asynchronous operation. It also validates the current operation.

And,

What it's also doing is introducing continuation, as we've mentioned earlier, the await keyword does the same behind the scene and puts all the code beneath it inside the continuation.

Also,

You can await the result of an async method that returns a Task because the method returns a Task, not because it's async. That means you can also await the result of a non-async method that returns a Task:

```
1 public async Task NewStuffAsync()
2 {
3   // Use await and have fun with the new stuff.
4   await ...
5 }
6
7 public Task MyOldTaskParallelLibraryCode()
8 {
9   // Note that this is not an async method, so we can't use await in here.
10   ...
11 }
12
```

```
13 public async Task ComposeAsync()
14 {
15   // We can await Tasks, regardless of where they come from.
16   await NewStuffAsync();
17   await MyOldTaskParallelLibraryCode();
18 }
```

Application Of Asynchronous Principles Accross .Net

Asynchronous principles are suited for any type of I/O operations. As we do in this case, we interact with an API over the web, but it could also be reading and writing from disk or memory or do things like database operations. In our case here, we're fetching some data from our API using the GetAsync method on our HttpClient.

The asynchronous principles that we talk about in our applications are not only meant for Windows applications or mobile applications. We can also apply the same principle to the server-side code in ASP.NET.

Awaiting In ASP.NET

Let's see an example:

```
1 using System;
2 using System.Net.Http;
3 using System.Threading.Tasks;
4 using System.Web.Http;
5
6 namespace WebApplication.Controllers
7 {
```

```
8   public class TestController : ApiController
9   {
10      // GET: api/Test
11      public async Task<IHttpActionResult> Get(string url)
12      {
13          var isValidUrl = Uri.TryCreate(url, UriKind.Absolute, out _);
14          if (!isValidUrl)
15          {
16              return BadRequest("Given url is not valid.");
17          }
18
19          var httpClient = new HttpClient();
20          var response = await httpClient.GetAsync(url).ConfigureAwait(false);
21          try
22          {
23
24              response.EnsureSuccessStatusCode();
25              var data = await response.Content.ReadAsStringAsync().ConfigureAwait(false);
26              return Ok(data);
27          }
28          catch (Exception ex)
```

```
29          {
30              return BadRequest(ex.Message);
31          }
32      }
33  }
34 }
```

Here is a test controller inside a web project that's allowing us to pretty much do the same thing that we do in our Windows application. However, we have a minor difference here:

```
1 .ConfigureAwait(false)
```

both of the task call this method in the end.

Why?

Avoiding Excessive Bouncing

When you await an async method, then it will capture the current "context" and later when the task completes, it will execute the remainder of the async method on a "context" that was captured before the "await" returned.

What Exactly Is That "Context"?

Simple answer:

If you're on a UI thread, then it's a UI context. If you're responding to an ASP.NET request, then it's an ASP.NET request context. Otherwise, it's usually a thread pool context.

Complex answer:

If SynchronizationContext.Current is not null, then it's the current SynchronizationContext. (UI and ASP.NET request contexts are SynchronizationContext contexts).

Otherwise, it's the current TaskScheduler (TaskScheduler.Default is the thread pool context). What does this mean in the real world? For one thing, capturing (and restoring) the UI/ASP.NET context is done transparently:

Why Capturing the Current Context is Needed?

One example of when it's necessary is in WPF apps. Imagine we delegate some kind of operation to another thread, and we want to use the result for setting a text box Text property. But the problem is, in this framework, only the thread that creates the UI element has the right to change its property. If we try to change a UI element from another thread, we get the InvalidOperationException error.

Most of the time, you don't need to sync back to the "main" context. You want to use ConfigureAwait(false) whenever the rest of that async method does not depend on the current context.

Also,

Some frameworks depending on their internal implementation don't need SynchronizationContext. Asp.Net Core is one such framework. Read more about it here. In short in such frameworks there might be no need for ConfigureAwait(false).

The Benefit Of Using Async And Await Inside ASP.NET?

Now what's interesting here is that this is not making the client-side app asynchronous.

So, what's the benefit?

The benefit of using async and await inside ASP.NET is to relieve IIS or the web server that you were using so that it can go ahead and work with other requests as your data is being loaded from disk, the database, or from another API. The primary benefit of asynchronous code on the server-side is scalability.

So as you notice here, the asynchronous principles are really powerful no matter if we are working in ASP.NET, in Windows, or any type of .NET applications.