**Chapter 6: The Threading Patterns**

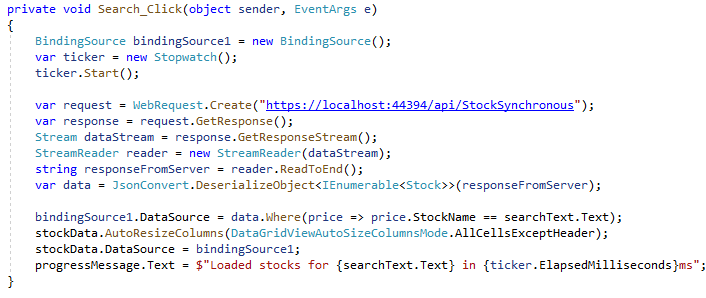
**Task-based Asynchronous Pattern (TAP)**

**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



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";

}

**Figure 6.1 – 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();

#region Async Calls

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;

#endregion

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

}

**Figure 6.2 – A button click event on a win form loading data from API asynchronously**

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

*Example here*

**Exception Handling**

**Progress reporting**

**Cancellation**

**Task.Run/FromResult/WhenAll/WhenAny/ContinueWith**

**Throttling**

**Retry**

**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/)