Applying Asynchronous Programming in C#

GETTING STARTED WITH ASYNCHRONOUS PROGRAMMING IN C# USING ASYNC AND AWAIT

Filip Ekberg
PRINCIPAL CONSULTANT & CEO

@fekberg fekberg.com

Works in Any .NET Application

WPF, WinForms, Xamarin

Console

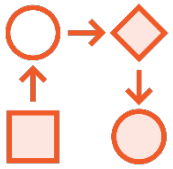
ASP.NET

Asynchronous Programming in .NET

Threading
(Low-level)

Background worker
(Event-based asynchronous pattern)

Task Parallel Library



Async and await

Synchronous vs Asynchronous

Synchronous

```
private void Search_Click(...)
{
    var client = new WebClient();
    var content =
            client.DownloadString(URL);


}
```

Asynchronous

```
private async void Search_Click(...)
{
    var client = new HttpClient();

    var response = await
            client.GetAsync(URL);

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

Asynchronous Web Request

```
private async void Search_Click(...)
{
    var client = new HttpClient();

    var response = await
            client.GetAsync(URL);

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

Asynchronous Web Request

```
private async void Search_Click(...)
{
    var client = new HttpClient();

    var response = await
            client.GetAsync(URL);

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

Asynchronous Web Request

```
private async void Search_Click(...)
{
    var client = new HttpClient();

    var response = await
            client.GetAsync(URL);

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

An **asynchronous operation** occurs in parallel and **relieves the calling thread** of the work

Setting up the Exercise Files

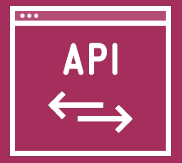
Ask questions on the discussion board

Introducing Async and Await in C#

Suited for I/O Operations

Disk

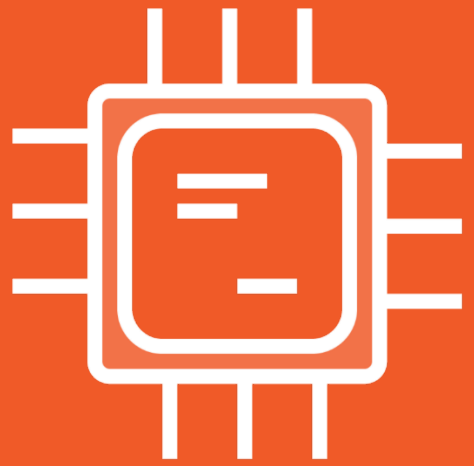
Memory



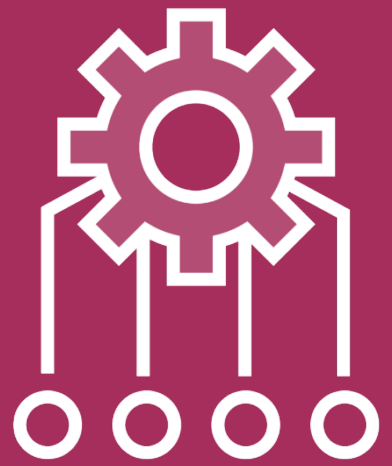
Web/API

Database

When to Use Parallel Programming



CPU bound operations



Independent chunks of data

An **asynchronous** operation **occurs in parallel**

Task Parallel Library

```
await Task.Run(() => {
    // I'm an asynchronous operation that is awaited
});


Parallel.Invoke(
    () => { /* Parallel Thread 1 */ },
    () => { /* Parallel Thread 2 */ },
    () => { /* Parallel Thread 3 */ },
    () => { /* Parallel Thread 4 */ },
);
```

Calling **Result** or **Wait()**
may cause a **deadlock**

Using **async** and **await** in **ASP.NET** means the **web server** can **handle other requests**

Obtaining the Result

```
Task<string> asynchronousOperation = GetStringAsync();

string result = await asynchronousOperation;
```

Using async and await

```
private async void Search_Click(...)
{
    var store = new DataStore();

    var responseTask = store.GetStockPrices("MSFT");

    var data = await responseTask;

    // Code below will run
    // when responseTask has completed

    Stocks.ItemsSource = data;
}
```

Always use **async** and
await together

Understanding a Continuation

Using async and await

```
private async void Search_Click(...)
{
    var store = new DataStore();

    var responseTask = store.GetStockPrices("MSFT");

    var data = await responseTask;

    // Code below will run
    // when responseTask has completed

    Stocks.ItemsSource = data;
}
```

The Await Keyword

Gives you a potential result

Validates the success of the operation

Continuation is back on calling thread

The **await** keyword introduces a **continuation**, **allowing** you to **get back to** the **original context** (thread)

Asynchronous Web Request

```
var response = await client.GetAsync(URL);
```

Continuation executed when GetAsync completes

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

Continuation executed when ReadAsStringAsync completes

```
var data = JsonConvert.DeserializeObject(...)
```

Creating Your Own Asynchronous Method

Implementing GetStocks()

Option 1:

Retrieve, process and return the stock data

Option 2:

Retrieve and process the stock data, then update the UI

Implementing GetStocks()

Option 1:

Retrieve, process and return the stock data

Option 2:

Retrieve and process the stock data, then update the UI

Only use **async void** for
event handlers

Handling an Exception

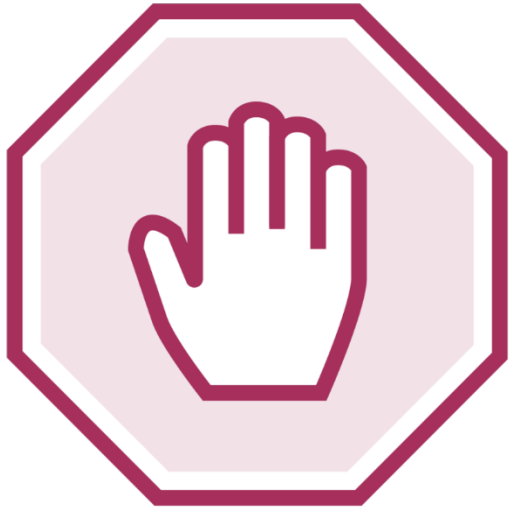
Introducing **asynchronous principles** can **improve** the **user experience**

Exceptions occurring
in an **async void** method
cannot be caught

Always use **await**
to **validate** your
asynchronous operations

Key Takeaways

Always **await** asynchronous operations



Avoid using **async void**

Best Practices

Using async & await

```
async Task Download()
{
    var client = new HttpClient();

    var response = await client.GetAsync(URL);

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

Using async & await

Task<HttpResponseMessage>

```
async Task Download()
{
    var client = new HttpClient();

    var response = await client.GetAsync(URL);

}
```

HttpResponseMessage

Using async & await

```
async Task Download()
{
    var client = new HttpClient();

    var response = await client.GetAsync(URL);

}
```

Validates the Task<HttpResponseMessage>
any exceptions will be re-thrown

Using async & await

```
async Task Download()
{
    var client = new HttpClient();

    var response = await client.GetAsync(URL);

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

Avoid using async void

```
async Task Good()
{
    throw new Exception("Find me on the Task");
}


async void Bad()
{
    throw new Exception("No one can catch me");
}
```

Unable to await

```
async Task Good()
{
    Bad(); // Can't await...


   // No way to run this line in a continuation
}

async void Bad()
{
    throw new Exception("No one can catch me");
}
```

Don't call
Result or **Wait()**

Deadlocking

```
private async void Search_Click(...)
{
    GetStocks().Wait();
}

private async Task GetStocks()
{
    ...
}
```

Causes a deadlock!

Using the result after await

```
private async void Search_Click(...)
{
    var store = new DataStore();

    var responseTask = store.GetStockPrices("MSFT");

    await responseTask;

    // In the continuation you may use Result
    var data = responseTask.Result;
}
```

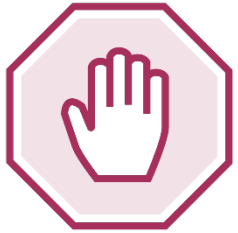
Best Practices

Always use async and await together

Always return a Task from an asynchronous method

Always await an asynchronous method to validate the operation

Use async and await all the way up the chain

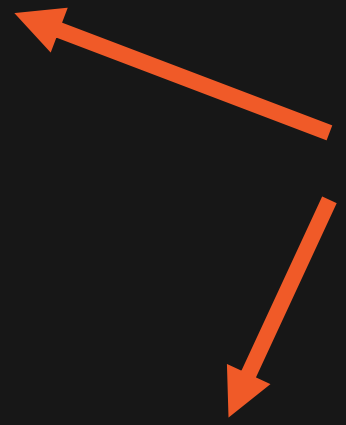


Never use async void unless it's an event handler or delegate

Never block an asynchronous operation by calling Result or Wait()

Different types of continuations

```
var response = await client.GetAsync(URL);
```



Very different continuations!

```
client.GetAsync(URL).ContinueWith((response) => {

});
```