FREE eBook

LEARNING async-await

Free unaffiliated eBook created from
Stack Overflow contributors.

#async-await

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: async-await

It is an unofficial and free async-await ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official async-await.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with async-await

Remarks

`async-await` allows asynchronous (means non-blocking, parallel) execution of code. It helps to keep your UI responsive at all times, while running potentially long operations in the background.

It is especially useful for I/O operations (like downloading from a server, or reading a file from the HDD), but it can also be used to execute CPU intensive calculations without freezing your application.

Examples

simple usage

Three things are needed to use `async-await`:

- The `Task` object: This object is returned by a method which is executed asynchronously. It allows you to control the execution of the method.
- The `await` keyword: "Awaits" a `Task`. Put this keyword before the `Task` to asynchronously wait for it to finish
- The `async` keyword: All methods which use the `await` keyword have to be marked as `async`

A small example which demonstrates the usage of this keywords

```
public async Task DoStuffAsync()
{
    var result = await DownloadFromWebpageAsync(); //calls method and waits till execution
finished
    var task = WriteTextAsync(@"temp.txt", result); //starts saving the string to a file,
continues execution right await
    Debug.Write("this is executed parallel with WriteTextAsync!"); //executed parallel with
WriteTextAsync!
    await task; //wait for WriteTextAsync to finish execution
}

private async Task<string> DownloadFromWebpageAsync()
{
    using (var client = new WebClient())
    {
        return await client.DownloadStringTaskAsync(new Uri("http://stackoverflow.com"));
    }
}

private async Task WriteTextAsync(string filePath, string text)
{
    byte[] encodedText = Encoding.Unicode.GetBytes(text);

    using (FileStream sourceStream = new FileStream(filePath, FileMode.Append))
    {
        await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
```

```
    }
}
```

Some things to note:

- You can specify a return value from an asynchronous operations with `Task<string>` or similar. The `await` keyword waits till the execution of the method finishes and returns the `string`.
- the `Task` object simply contains the status of the execution of the method, it can be used as any other variable.
- if an exception is thrown (for example by the `WebClient`) it bubbles up at the first time the `await` keyword is used (in this example at the line `var result (...)`)
- It is recommended to name methods which return the `Task` object as `MethodNameAsync`

execute synchronous code asynchronous

If you want to execute synchronous code asynchronous (for example CPU extensive calculations), you can use `Task.Run(() => {})`.

```
public async Task DoStuffAsync()
{
    await DoCpuBoundWorkAsync();
}


private async Task DoCpuBoundWorkAsync()
{
    await Task.Run(() =>
    {
        for (long i = 0; i < Int32.MaxValue; i++)
        {
            i = i ^ 2;
        }
    });
}
```

the Task object

The `Task` object is an object like any other if you take away the `async-await` keywords.

Consider this example:

```
public async Task DoStuffAsync()
{
    await WaitAsync();
    await WaitDirectlyAsync();
}

private async Task WaitAsync()
{
    await Task.Delay(1000);
}

private Task WaitDirectlyAsync()
{
```

```
    return Task.Delay(1000);
}
```

The difference between this two methods is simple:

- `WaitAsync` wait for `Task.Delay` to finish, and then returns.
- `WaitDirectlyAsync` does not wait, and just returns the `Task` object instantly.

Every time you use the `await` keyword, the compiler generates code to deal with it (and the `Task` object it awaits).

- On calling `await WaitAsync()` this happens twice: once in the calling method, and once in the method itself.
- On calling `await WaitDirectlyAsync` this happens only once (in the calling method). You would therefore archive a little speedup compared with `await WaitAsync()`.

Careful with `exceptions`: `Exceptions` will bubble up the first time a `Task` is `await`ed. Example:

```
private async Task WaitAsync()
{
    try
    {
        await Task.Delay(1000);
    }
    catch (Exception ex)
    {
        //this might execute
        throw;
    }
}

private Task WaitDirectlyAsync()
{
    try
    {
        return Task.Delay(1000);
    }
    catch (Exception ex)
    {
        //this code will never execute!
        throw;
    }
}
```

async void

You can use `void` (instead of `Task`) as a return type of an asynchronous method. This will result in a "fire-and-forget" action:

```
public void DoStuff()
{
    FireAndForgetAsync();
}
```

```
private async void FireAndForgetAsync()
{
    await Task.Delay(1000);
    throw new Exception(); //will be swallowed
}
```

As you are returning `void`, you can not `await FireAndForgetAsync`. You will not be able to know when the method finishes, and any exception raised inside the `async void` method will be swallowed.

Read Getting started with async-await online: https://riptutorial.com/async-await/topic/5658/getting-started-with-async-await

Chapter 2: Best practices

Examples

Avoid async void

- The only place where you can safely use `async void` is in event handlers. Consider the following code:

```
private async Task<bool> SomeFuncAsync() {
  ...
  await ...
}
public void button1_Click(object sender,  EventArgs e) {
  var result = SomeFuncAsync().Result;
  SomeOtherFunc();
}
```

  Once the `async` call completes, it waits for the `SynchronizationContext` to become available. However, the event handler holds on to the `SynchronizationContext` while it is waiting for `SomeFuncAsync` method to complete; thus causing a circular wait (deadlock).

  To fix this we need to modify the event handler to:

```
public async void button1_Click(object sender,  EventArgs e) {
  var result = await SomeFuncAsync();
  SomeOtherFunc();
}
```

- Any exception thrown out of an `async void` method will be raised directly on the `SynchronizationContext` that was active when the `async void` method started.

```
private async void SomeFuncAsync() {
  throw new InvalidOperationException();
}
public void SomeOtherFunc() {
  try {
    SomeFuncAsync();
  }
  catch (Exception ex) {
    Console.WriteLine(ex);
    throw;
  }
}
```

  the exception is never caught by the catch block in `SomeOtherFunc`.

- `async void` methods don't provide an easy way to notify the calling code that they've completed

- `async void`

methods are difficult to test. MSTest asynchronous testing support only works for `async` methods returning `Task` or `Task<T>`.

Read Best practices online: https://riptutorial.com/async-await/topic/9055/best-practices

Credits

S. No	Chapters	Contributors
1	Getting started with async-await	Community, Florian Moser, Kirill Mehtiev
2	Best practices	user2321864