

## ESTA COUVE FLOR



# Introdução a async e await

### SUMÁRIO

Introduzindo as palavras-chaves

Awaitables

Tipos de retorno

Retornando valores

Contexto

Uma resposta simples:

Resposta complexa:

Evitando o Contexto

Async Composition

Orientações

Próximos passos

*Este artigo é uma tradução e adaptação autorizada do original [“Async and Await”](#), de autoria do MVP Stephen Cleary.*

## Introduzindo as palavras-chaves

Irei usar alguns conceitos que serão explicados posteriormente, por isso, acalme-se e concentre-se nesta primeira parte.

Métodos assíncronos tem a seguinte aparência:

```
1 public async Task DoSomethingAsync()
```

A palavra-chave **async** habilita a palavra-chave **await** naquele método e modifica como o resultado é tratado. E isso é tudo que *async* faz! Não faz com que ele rode em um *thread pool* ou algum tipo de magia negra, a palavra-chave *async* apenas habilita a palavra-chave *await* (e gerencia o retorno dos métodos).

O início de um método *async* é executado como qualquer outro, ou seja, roda de forma **síncrona** até que ele alcance o primeiro *await* (ou lance uma exceção).

A palavra-chave **await** é onde as coisas se tornam assíncronas. *await* é como um **operador unário**: Ele recebe um argumento simples, um *awaitable* (uma operação assíncrona) e o examina para ver se ele já completou; se ele estiver completado, o método apenas continua (de forma síncrona) como um método normal, mas se o *await* notar que o *awaitable* ainda não completou, ele atua de forma assíncrona, pedindo para o *awaitable* rodar o restante do método até que se complete e **devolve à execução para quem chamou o método *async***.

Mais tarde, quando o *awaitable* completar, ele executará o restante do método *async*. Se você usar a palavra-chave *await* em um *awaitable* padrão (tal como *Task*), o restante do método *async* irá executar no **contexto** que foi capturado antes do *await* ter sido retornado.

Gosto de pensar no *await* como *asynchronous wait* (espera assíncrona). Isso quer dizer que o método *async* **pausa** até que o *awaitable* complete (então ele espera), mas a *thread* atual não é **bloqueada** (por isso ele é assíncrono);

## Awaitables

Como eu mencionei *await* recebe um argumento – um *awaitable* – que é uma operação assíncrona. Existem dois tipos disponíveis de *awaitables* no Framework .NET: *Task<T>* e *Task*.

Há também outros *awaitable*: métodos especiais tais como *Task.Yield* que retornam *awaitables* que não são *Tasks* e o *Runtime WinRT* que possui um *awaitable* não gerenciado. Você também pode criar o seu próprio (geralmente por razões de

performance) ou usar métodos de extensão para fazer um tipo que não é *awaitable* se tornar um. Isso é tudo que direi sobre como criar os seus próprios *awaitables*, já que em todo o tempo que eu usei *async/await* poucas vezes tive que escrever um para mim. Se você quiser saber mais sobre como escrever um personalizado, veja o [blog do time de parallel](#) e o [blog do Jon Skeet](#).

Um ponto importante sobre os *awaitable* é: **O tipo que é *awaitable* e não o método**. Em outras palavras, você pode usar o *await* em um resultado de um método *async* que retorna uma *Task*... Porque o método retorna uma *Task*, *não* por que ele é *async*. Logo, você também pode usar o *await* no resultado de um método não assíncrono que retorna uma *Task*.

```
1    public async Task NewStuffAsync()
2    {
3        // Use await e se divirta com a coisa nova
4        await ...
5    }
6
7    public Task MyOldTaskParallelLibraryCode()
8    {
9        // Note que este método não é async então não podemos usar await aqui
10       ...
11    }
12
13    public async Task ComposeAsync()
14    {
15        // Nós podemos chamar o await em Tasks, não importando de onde elas viera
16        await MyOldTaskParallelLibraryCode();
17    }
```

**Dica:** Se você tem um método assíncrono muito simples, você pode escrevê-lo sem usar a palavra-chave *await* (ex. usando *Task.FromResult*). Se você puder escrever sem, **faça** e também remova a palavra chave *async*. Um método não *async* que retorna um *Task.FromResult* é mais eficiente que um método *async* que retorna um valor.

## Tipos de retorno

Métodos *async* podem retornar *Task<T>*, *Task* e *void*. Na maioria dos casos, você deverá retornar um *Task<T>* ou *Task* e **void** apenas quando **realmente** for necessário.

Por que retornar *Task<T>* ou *Task*? Por que eles são *awaitables* e *void* **não**. Se você

tem um método assíncrono que retorna uma *Task*, você poderá passar o resultado para um *await*, já com o *void*, você não terá nada o que passar, em resumo, você só deve usar *void* quando estiver escrevendo *event handlers* assíncronos.

**Dica:** Você também pode usar *async void* para outros tipos de ações, ex. um método único do tipo *static async void MainAsync()* para aplicações de console. Porém, este uso do *async void* tem seus próprios problemas; veja [Async Console Programs](#). O caso de uso principal para métodos *async void* é para *event handlers* onde a conclusão/retorno do método não é importante.

## Retornando valores

Métodos assíncronos retornando *Task* ou *void* não possuem um valor de retorno, já os métodos que retornam *Task<T>* deve retornar um valor do tipo *T*.

```
1    public async Task<int> CalculateAnswer()
2    {
3        await Task.Delay(100); // (Probably should be longer...)
4
5        // Retorna um "int" não uma "Task<int>"
6        return 42;
7    }
```

É um pouco estranho de se acostumar, mas há boas razões por trás deste design.

## Contexto

Em nossa introdução, eu mencionei que quando você usa a palavra-chave *await* em um tipo *awaitable* padrão, ele irá capturar o “contexto” e depois aplicá-lo ao restante do método, mas o que exatamente é este “Contexto”?

### Uma resposta simples:

- Se você está na *thread de UI*, então o contexto será o de UI.
- Se você estiver espondendo a uma requisição ASP.NET, então ele será o contexto da requisição.
- Caso contrário, geralmente será o contexto do *thread pool*.

### Reposta complexa:

- Se o *SynchronizationContext.Current* não for *null*, então ele será o atual *SynchronizationContext*. (Os contexto de UI e requisição do asp.net são *SynchronizationContexts*)
- Caso contrário, será o *TaskScheduler* atual (*TaskScheduler.Default* é o contexto do thread pool). O que isso quer dizer no mundo real? Uma coisa, a captura e recuperação e dos contextos de UI/ASP.NET são feitos de forma transparente.

```
1      // Exemplo de WinForms (Funciona do mesmo jeito para WPF).
2      private async void DownloadFileButton_Click(object sender, EventArgs e)
3      {
4          // Uma vez que a gente espera assíncronamente, a thread de UI não é bloc
5          await DownloadFileAsync(fileNameTextBox.Text);
6
7          // Após a conclusão, nós retomamos no context de UI e podemos acessar i
8          resultTextBox.Text = "File downloaded!";
9      }
10
11     // Exemplo em ASP.NET
12     protected async void MyButton_Click(object sender, EventArgs e)
13     {
14         // Já que a gente está esperando de forma assíncrona, a thread do ASP.N
15         // isso permite que ela fique disponível para executar outros requests
16         await DownloadFileAsync(...);
17
18         // Depois de finalizado, retomamos o context e podemos acessar os dados
19         // Nós poderemos estar em outra thread, mas nós teremos o mesmo context
20         Response.Write("File downloaded!");
21     }
```

Isso é ótimo para *event handlers*, mas nem sempre será o que você irá precisar para maioria do código (que será boa parte código assíncrono que você escreverá).

## Evitando o Contexto

Na maior parte do tempo, você não precisa voltar ao contexto principal. Boa parte dos métodos assíncronos são projetados com composição em mente: Eles usam o *await* de outras operações, cada um representando uma operação assíncrona por si mesma (podendo ser compostas por outras). Neste caso, você pode dizer ao *awaiter* para não capturar o contexto atual chamando *ConfigureAwait* e passando *false* para ele. Exemplo:

```
1      private async Task DownloadFileAsync(string fileName)
```

```

2      {
3          // Use HttpClient or whatever to download the file contents.
4          var fileContents = await DownloadFileContentsAsync(fileName).ConfigureAwait(
5
6          // Note that because of the ConfigureAwait(false), we are not on the o
7          // Instead, we're running on the thread pool.
8
9          // Write the file contents out to a disk file.
10         await WriteToDiskAsync(fileName, fileContents).ConfigureAwait(false);
11
12         // The second call to ConfigureAwait(false) is not *required*, but it i
13     }
14
15     // WinForms example (it works exactly the same for WPF).
16     private async void DownloadFileButton_Click(object sender, EventArgs e)
17     {
18         // Since we asynchronously wait, the UI thread is not blocked by the f
19         await DownloadFileAsync(fileNameTextBox.Text);
20
21         // Since we resume on the UI context, we can directly access UI element
22         resultTextBox.Text = "File downloaded!";
23     }

```

O que é importante notar com este exemplo é cada *nível* de chamada de um método assíncrono tem seu próprio contexto. *DownloadFileButton\_Click* inicia no contexto da UI, chama *DownloadFileAsync*. *DownloadFileAsync* também inicia no contexto da UI, mas então o deixa para trás ao chamar *ConfigureAwait(false)*. O resto do método *DownloadFileAsync* executa no contexto da *thread pool*. Porém quando *DownloadFileAsync* completa, o *DownloadFileButton\_Click* continua, ele, de fato, continua no contexto da UI.

Uma boa regra a seguir é de sempre usar *ConfigureAwait(false)* a menos que você saiba que realmente irá precisar do contexto.

## Async Composition

Até agora nós apenas consideramos composição serial: um método assíncrono aguarda por uma operação a cada tempo, mas também é possível iniciar várias operações e esperar por uma (ou todas) completarem. Você pode fazer isso iniciando operações, mas não usando *await* imediatamente.

```

1     public async Task DoOperationsConcurrentlyAsync ()
2     {
3         Task[] tasks = new Task[3];

```

```
4      tasks[0] = DoOperation0Async();
5      tasks[1] = DoOperation1Async();
6      tasks[2] = DoOperation2Async();
7
8      // Neste ponto, todas as três tarefas estão rodando ao mesmo tempo.
9      // Agora a gente await todas elas de uma vez.
10     await Task.WhenAll(tasks);
11 }
12
13 public async Task<int> GetFirstToRespondAsync()
14 {
15     // Chama dois web services; Pega a primeira resposta
16     Task<int>[] tasks = new[] { WebService1Async(), WebService2Async() };
17
18     // Await pelo primeiro a responder
19     Task<int> firstTask = await Task.WhenAny(tasks);
20
21     // Retorna o resultado
22     return await firstTask;
23 }
```

Ao usar composição concorrente (`Task.WhenAll` ou `Task.WhenAny`), você pode executar operações concorrentes simples. Você também pode usar estes métodos junto com `Task.Run` para fazer computações simples em paralelo. Mas isso não é um substituto para a `Task Parallel Library` – qualquer operação que faça uso intensivo de CPU e operações paralelas devem fazer através da TPL.

## Orientações

Leia o documento da [Task-based asynchronous \(TAP\)](#). Ele é extremamente bem escrito e inclui informações sobre o design da api e uso apropriado das palavras chaves *async* e *await* assim como inclusão de eventos de cancelamento e progresso.

Existem muitas novas técnicas compatíveis com *await* que devem ser usadas ao invés das técnicas antigas onde há bloqueio da aplicação. Se você tem alguns desses antigos exemplos no seu código novo com *async*, você está fazendo isso errado:

Antigo	Novo	Descrição
<code>task.Wait</code>	<code>await task</code>	Aguarda/await uma task completar.
<code>Task.Result</code>	<code>await task</code>	Pega o resultado de uma task completada
<code>Task.WaitAny</code>	<code>await Task.WhenAny</code>	Aguarda/await que uma de uma coleção de tasks complete.

Antigo	Novo	Descrição
Task.WaitAll	await Task.WhenAll	Aguarda/await que todas as tasks de uma coleção complete
Thread.Sleep	await Task.Delay	Aguarda/await por um periodo de tempo.
Task constructor	Task.Run or TaskFactory.StartNew	Cria uma task baseada em código

## Próximos passos

Stephen Cleary publicou um artigo na MSDN intitulado [Best practices in Asynchronous Programming](#) (Melhores práticas em programação assíncrona) que explica de forma mais completa as orientações sobre “evite void async”, “assíncrono por todo caminho” e “configure o contexto”. A [documentação oficial da MSDN](#) é muito boa; Eles incluem uma versão online do documento [Task-based Asynchronous Pattern](#) que é excelente e cobre os designs dos métodos assíncronos. A equipe *async* publicou uma [faq sobre async/await](#), um grande lugar para continuar aprendendo sobre *async*. Eles possuem destaques dos melhores posts e vídeos por lá. E também qualquer [blog post do Stephen Toub](#) é bem instrutivo. E claro, outro recurso é o blog do [Stephen Cleary](#). Todo este material em inglês.

---

### COMPARTILHE



Introdução a *async* e *await* publicado em June 26, 2016 e modificado em June 26, 2016.



**2 Comments**   **Esta couve flor****1 Login** ▾ **Recommend** 1 **Tweet** **Share****Sort by Best** ▾

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS **Jaedson** • 2 years ago

Ajudou muito, agora estou entendendo melhor o que o CodeCracker diz.

 |  • [Reply](#) • [Share](#) ›**Fabio Henrique Gabriele** • 2 years ago

Muito bom o artigo

 |  • [Reply](#) • [Share](#) › **Subscribe**    **Add Disqus to your site**[Add Disqus](#)[Add](#)[Disqus](#) [Disqus](#) [Disqus](#) [Disqus](#) [Disqus](#) [Disqus](#)**NÃO DEIXE DE VER TAMBÉM****(VER TODAS AS PUBLICAÇÕES)**

- [Covariância e contravariância no C Sharp](#)
- [Meu novo desafio: Aprender F Sharp](#)
- [Por que continuar com stack .NET da Microsoft](#)

© 2017 Cleiton Loiola. Powered by Jekyll using the Minimal Mistakes theme.