

Async / Await Pattern

Task-based Asynchronous

Leandro Silva, SET/21



pricefy

pricefyLabs

<https://medium.com/pricefy-labs>

Estamos contratando para os
times de engenharia e
produtos.

vamos conversar?



ProdOps - Engenharia e Produto com Leandro Silva

<https://www.youtube.com/watch?v=2lxX2f0ZckQ>

<https://www.youtube.com/watch?v=jOeuK2U8vI8>



**ElvenWorks - Conhecendo a tecnologia por trás de uma
solução muito inteligente de Precificação**

<https://www.youtube.com/watch?v=DCTOI0RcrUo>

async/await 101

É um recurso sintático disponível em diversas linguagens

assíncrono e não-bloqueante

Código **assíncrono**
escrito como se fosse
síncrono

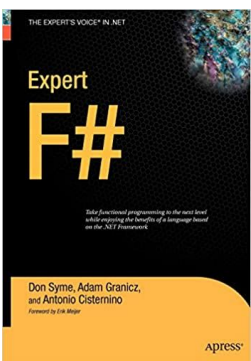
(bye bye callback hell)

Semanticamente
similar às **corrotinas**

(execute → suspende → resume)

Certo código pode
executar **enquanto**
aguarda por outro
código dispendioso

(single-thread state machine)



Introducing Asynchronous Computations

The two background worker samples we've shown so far run at "full throttle." In other words, the computations run on the background threads as active loops, and their reactive behavior is limited to flags that check for cancellation. In reality, background threads often have to do different kinds of work, either by responding to completing asynchronous I/O requests, by processing messages, by sleeping, or by waiting to acquire shared resources. Fortunately, F# comes with a powerful set of techniques for structuring asynchronous programs in a natural way. These are called asynchronous workflows. In the next three sections, we cover how to use asynchronous workflows to structure asynchronous and message-processing tasks in ways that preserve the essential logical structure of your code.

F# 1.9.2.9, 2007

International Symposium on Practical Aspects of Declarative Languages
PADL 2011: [Practical Aspects of Declarative Languages](#) pp 175-189 | [Cite as](#)

The F# Asynchronous Programming Model

Authors Authors and affiliations

Don Syme, Tomas Petricek, Dmitry Lomov

Conference paper



Part of the [Lecture Notes in Computer Science](#) book series (LNCS, volume 6539)

Abstract

We describe the asynchronous programming model in F#, and its applications to reactive, parallel and concurrent programming. The key feature combines a core language with a non-blocking modality to author lightweight asynchronous tasks, where the modality has control flow constructs that are syntactically a superset of the core language and are given an asynchronous semantic interpretation. This allows smooth transitions between synchronous and asynchronous code and eliminates callback-style treatments of inversion of control, without disturbing the foundation of CPU-intensive programming that allows F# to interoperate smoothly and compile efficiently. An adapted version of this approach has recently been announced for a future version of C#.

Task asynchronous programming model

08/19/2020 • 13 minutes to read •

You can avoid performance bottlenecks and enhance the overall responsiveness of your application by using asynchronous programming. However, traditional techniques for writing asynchronous applications can be complicated, making them difficult to write, debug, and maintain.

C# 5 introduced a simplified approach, async programming, that leverages asynchronous support in the .NET Framework 4.5 and higher, .NET Core, and the Windows Runtime. The compiler does the difficult work that the developer used to do, and your application retains a logical structure that resembles synchronous code. As a result, you get all the advantages of asynchronous programming with a fraction of the effort.

C# 5.0, 2012

Async/Await for ES6 targets

With the 1.7 release, TypeScript now supports [Async functions](#) for targets that have [ES6 generator](#) support enabled (e.g. node.js v4 and above). Functions can now be prefixed with the `async` keyword designating it as an asynchronous function. The `await` keyword can then be used to stop execution until an `async` function's promise is fulfilled. Following is a simple example:

TypeScript 1.7, 2015

Async-await on stable Rust!

Nov. 7, 2019 • Niko Matsakis

So, what is async await? Async-await is a way to write functions that can "pause", return control to the runtime, and then pick up from where they left off. Typically those pauses are to wait for I/O, but there can be any number of uses.

You may be familiar with the async-await from JavaScript or C#. Rust's version of the feature is similar, but with a few key differences.

Rust 1.39.0, 2019

Não é sobre paralelização ou concorrência per se i/o assíncrono do jeito certo

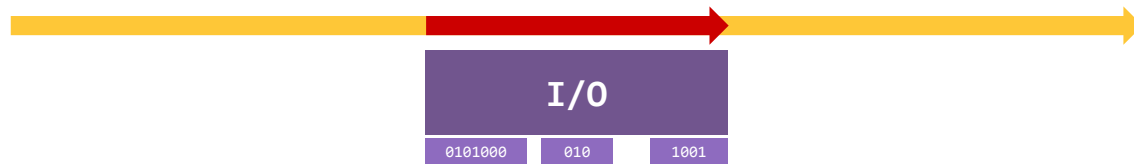
Task-based Asynchronous Programming não é sobre paralelização de rotinas computacionais síncronas ou concorrência por recursos compartilhados. Existe outras técnicas para isso.

Na verdade, ela existe para quando se precisa de concorrência e reatividade em **rotinas dependentes de I/O**, onde não se quer desperdiçar um número de threads, bloqueadas, sem fazer nada, enquanto esperam por respostas de I/O para seguir seu curso de vida.

Síncrono

Bloqueada, aguardando...

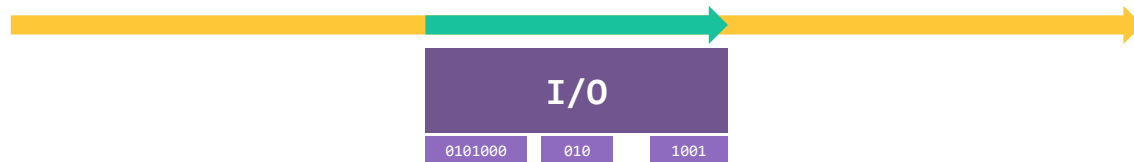
Thread



Assíncrono

Fazendo outra coisa, enquanto aguarda...

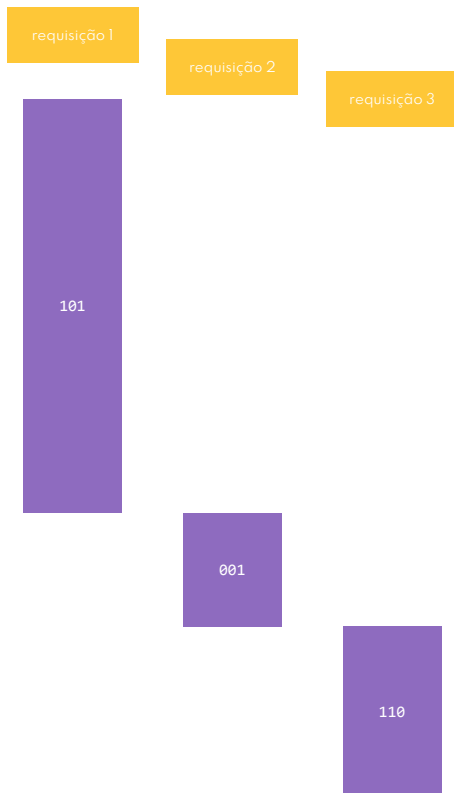
Thread



Tempo

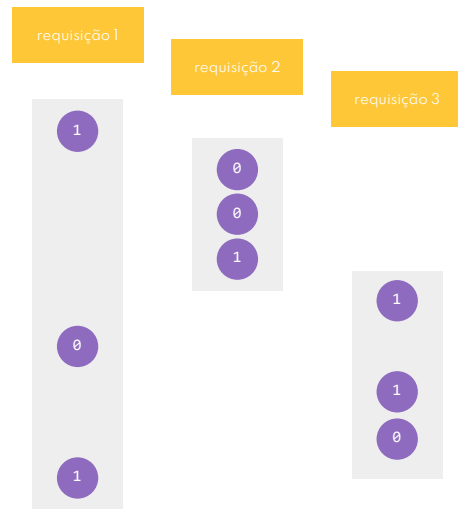
Thread

I/O Bloqueante



Thread

I/O Não-Bloqueante



THREADS NÃO SÃO DE GRAÇA

O custo de cada thread só é irrelevante quando se tem poucas


use com moderação

- 1 megabyte em user space;
- 12 kilobytes (32 bits) ou 24 kilobytes (64 bits) em kernel space;
- Tempo para alocar, inicializar e, depois, liberar memória;
- Attach/Detach Notification para cada DLL do processo (centenas delas);
- Context switch entre threads **a cada 30 ms** (porque cada **core** só executa uma por vez).

O grande poder do estilo de programação **async/await** está em poder liberar, para fazer qualquer outra coisa, a thread que fez a chama, enquanto ela **aguarda** uma operação de I/O ser concluída.

melhor utilização de threads

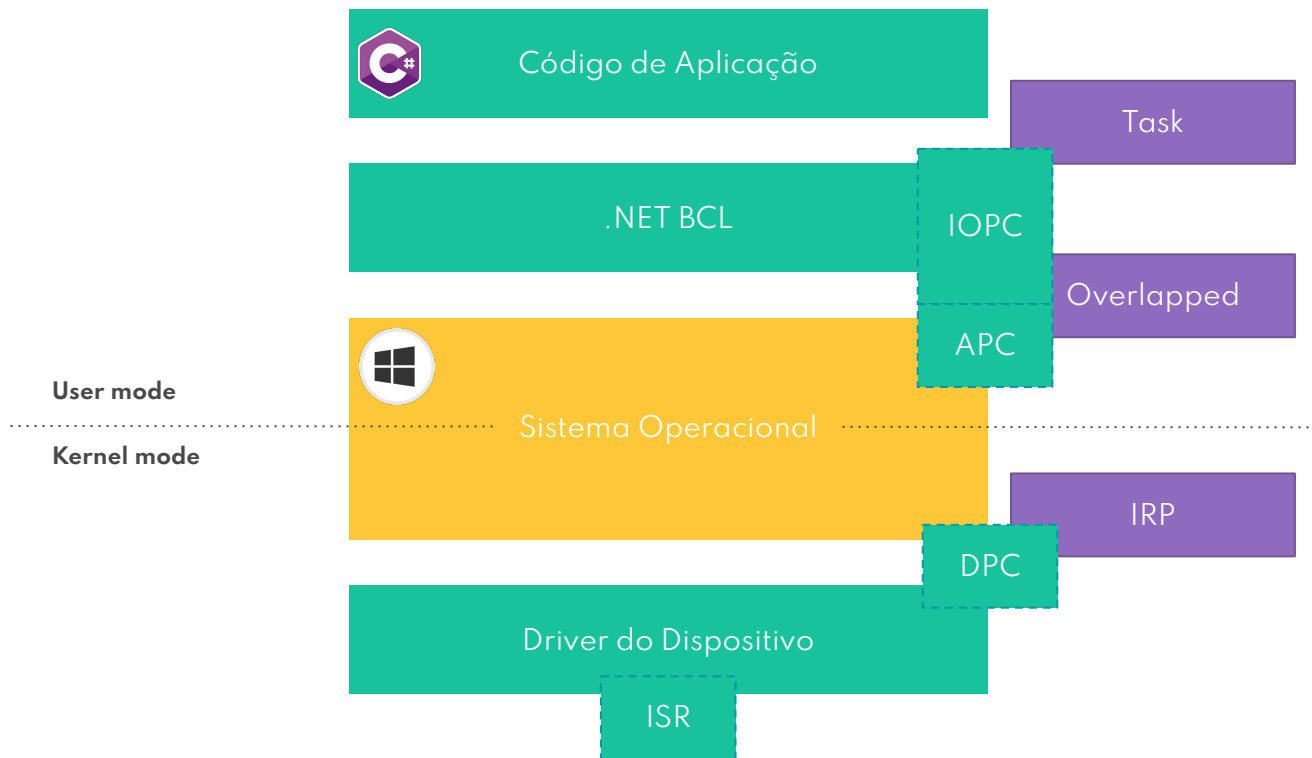
Um menor número de threads é capaz de lidar com a mesma quantidade de operações, se compararmos os modelos **blocking** vs **non-blocking**.
melhor escalabilidade



O número máximo de threads está diretamente relacionado à quantidade de memória virtual disponível - quando todas estão ocupadas, tarefas aguardam na fila.

thread pool 101

BCL - Base Class Library
IOPC - I/O Completion Port
APC - Asynchronous Procedure Call
DPC - Deferred Procedure Call
ISR - Interrupt Service Routine
Task - System.Threading.Tasks
Overlapped - Win32 Overlapped I/O
IRP - I/O Request Packet



IOCP é um mecanismo super eficiente de async I/O no Windows. Algumas poucas threads (normalmente, uma por processador) monitoram uma gigantesca quantidade de operações em uma única "porta".

Em **Linux**, **epoll** é usada em lugar de **IOCP**.

Async? **Como?**

```
1 function callbackHell() {  
2   doSomething1(function () {  
3     doSomething2(function () {  
4       doSomething3(function () {  
5         doSomething4(function () {  
6           doSomething5(function () {  
7             doSomething6(function () {  
8               doSomething7(function () {  
9                 doSomething8(function () {  
10                  doSomething9(function () {  
11                   // ...  
12                  });  
13                 });  
14                });  
15               });  
16              });  
17             });  
18            });  
19           });  
20          });  
}
```

Antes: Callback Hell

(continuation-passing style)



Agora: Async / Await

Em essência, é uma maneira mundana de encadear (**bind**) uma série de unidades sintáticas (**statements**), plugando nossa própria lógica entre elas.

computational expression

(shhh! don't say that **m-word**)

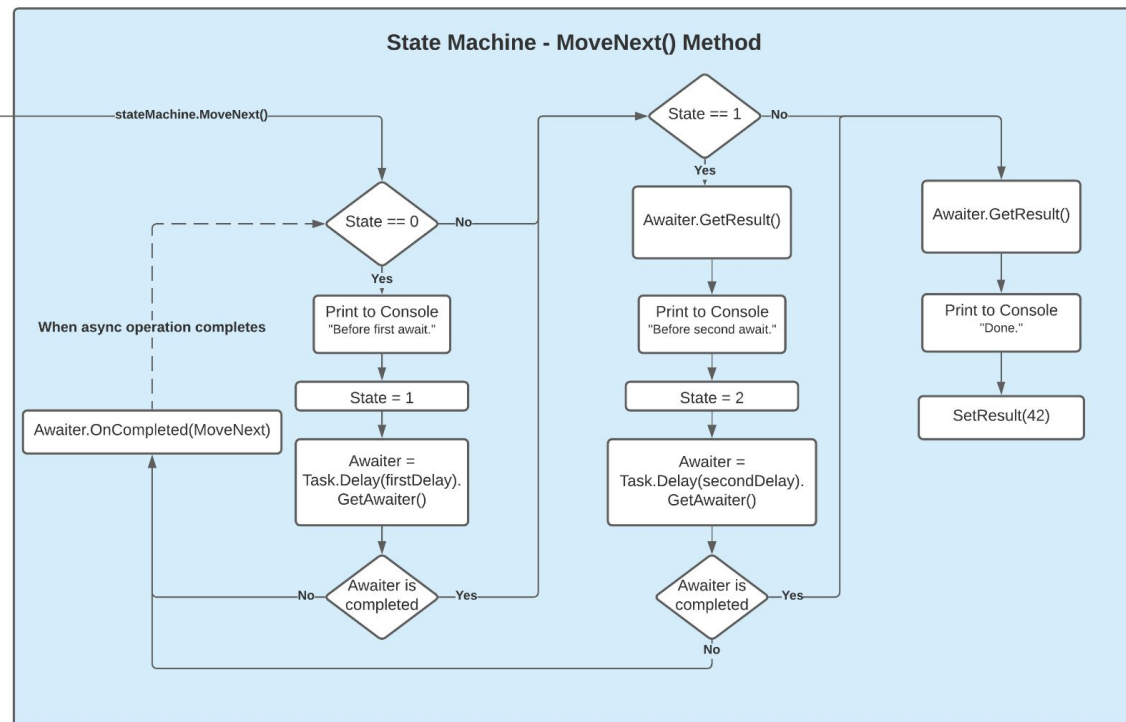
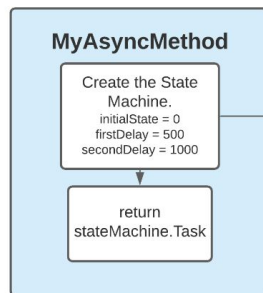
```
using System;
using System.Threading.Tasks;

var content = await GetContentAsync("https://medium.com/pricefy-labs");
Console.WriteLine(content);

var success = await SaveAsync(content);
Console.WriteLine(success);

static async Task<string> GetContentAsync(string url) ...

static async Task<bool> SaveAsync(string content) ...
```

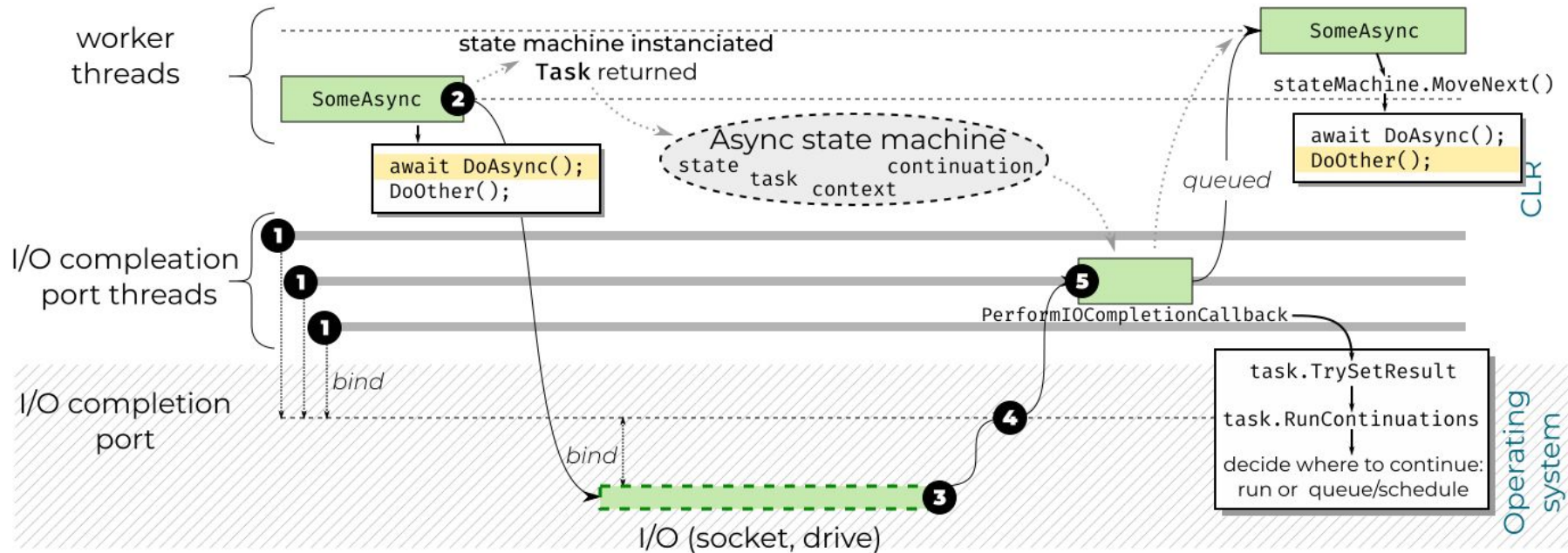
Por baixo dos panos

await marca um checkpoint na execução de um método **async**, onde pode ser que ele seja pausado (ou seja, retorne uma **Task** incompleta), caso a operação assíncrona ainda não tenha sido completada, e então, certo tempo depois, tendo completado, retorne em uma "continuation".

```

public static async Task<int> MyAsyncMethod(int firstDelay, int secondDelay)
{
    Console.WriteLine("Before first await.");           State = 0
    await Task.Delay(firstDelay);
    Console.WriteLine("Before second await.");         State = 1
    await Task.Delay(secondDelay);
    Console.WriteLine("Done.");                         State = 2
    return 42;
}
  
```

2+1



- 1 IOCP threads are blocked for notification on assigned IOCP
- 2 `DoAsync` from worker thread "opens" device, "binds" it to IOCP and starts overlapped I/O - **state machine** is created and **Task** returned
- 3 I/O operation completes

- 4 IOCP is signalled
- 5 one of the blocked IOCP threads is signalled and runs **state machine** to set **Task** result - deciding when to run a **continuation**: inlined, `SynchronizationContext`, `TaskScheduler`, ...

Aprendendo com **erros comuns**

Tarefas não **awaited** escondem exceções

- **Fire&Forget** sem o tratamento devido, não dá ao compilador a chance de gerar um **try/catch** para o **MoveNext**;
- Quando acontece uma exceção, você não tem como capturar;
- Não use **.Wait()** ou **.Result** para resolver isso, porque são síncronos e lançam **AggregateException**;
- **GetAwaiter().GetResult()** resolve, você captura a sua exceção, mas assim como **.Wait()** ou **.Result**, também é síncrono;
- Use **await**!

Evite deadlocks, não `.Wait()` ou `.Result`

- Quando você `.Wait()` ou `.Result` uma **Task**, você torna a chamada síncrona e bloqueia a thread atual;
- Isso não é seguro, tem risco de deadlock, dependendo de onde a **Task** foi agendada **SynchronizationContext** ou **ThreadScheduler**;
- Se for uma chamada em um construtor, use um **factory method** assíncrono em seu lugar;
- Se for um método síncrono de uma interface, que você precisa implementar (e.g. **IDisposable**), use **ConfigureAwait(false)** na **Task** e então **GetAwaiter().GetResult()** ela;
- Em qualquer outra situação, use **await**!

Nunca declare métodos **async void**

Uma exceção à regra são os **event handlers** usados em aplicações Windows Forms ou WPF, porque eles precisam ter uma assinatura específica, que por acaso, retorna **void**.

- Não é possível **await** um método **async void**;
- Um **try/catch** em sua chamada não captura exceções;
- Exceções acabam acontecendo em um outro contexto;
- Use **async Task** e **await** ele!
- Se tiver que usar em uma lambda, que essa lambda seja uma **Func<Task<T>>** e nunca uma **Action<T>**, que seria **void** no final.

Não use `async Task`, se não precisar `await`

- Se tudo que o método faz, após uma lógica qualquer, é `await` um outro método e `return` seu resultado, remova o `async` da assinatura e simplesmente `return` sua `Task<T>` - a menos que o `return` esteja em um bloco `try/catch`;
- Se for um valor constante ou computado sincronamente, retorne `Task.FromResult` ou uma `ValueTask`, sem `await`;
- Isso evita que o compilador gere uma máquina de estado para o método só porque foi anotado como `async`.

Sempre passe um **CancellationToken**

- Você não vai querer continuar uma operação custosa que já foi cancelada, vai?
- Em web apps, por exemplo, um request pode ser cancelado;
- Obtenha um **CancellationToken** na action e passe ele adiante;
- Não se esqueça de capturar a **TaskCanceledException** que será lançada.

Fire&Forget `Task<T>` de modo seguro

- Uma das coisas mais importantes em um fire&forget seguro é fazer uso de `ConfigureAwait(false)` na `Task`;
- A outra coisa é evitar que uma exceção não capturada crash a aplicação - aliás, sempre `await` dentro de um bloco `try/catch`;
- Opcionalmente, ofereça a possibilidade de ser notificado de alguma exceção via callback.

Se aprofundando
no assunto

Async I/O & Threads

- Performing Asynchronous I/O Bound Operations (Jeffrey Richter)
<https://youtu.be/hBOKIJWFoqs>
- Pushing the Limits of Windows: Processes and Threads
<https://techcommunity.microsoft.com/t5/windows-blog-archive/pushing-the-limits-of-windows-processes-and-threads/ba-p/723824>
- There Is No Thread
<https://blog.stephencleary.com/2013/11/there-is-no-thread.html>
- I/O Completion Ports
<https://docs.microsoft.com/en-us/windows/win32/fileio/i-o-completion-ports>

Async Programming Foundation

- Async/await
<https://en.wikipedia.org/wiki/Async/await>
- Introducing F# Asynchronous Workflows
<https://docs.microsoft.com/en-us/archive/blogs/dsyme/introducing-f-asynchronous-workflows>
- Computation expressions: Introduction
<https://fsharpforfunandprofit.com/posts/computation-expressions-intro/#series-toc>
- Computation Expressions in C# using async/await
<https://blog.neteril.org/blog/2018/01/11/computation-expressions-csharp-async-await/>

Async C# The Right Way

- .NET async/await in a single picture
<https://tooslowexception.com/net-asyncawait-in-a-single-picture/>
- Exploring the async/await State Machine – Main Workflow and State Transitions
<https://vkontech.com/exploring-the-async-await-state-machine-main-workflow-and-state-transitions/>
- Correcting Common Async/Await Mistakes in .NET
<https://codetraveler.io/ndcoslo-asyncawait/>
- C# Async Antipatterns
<https://markheath.net/post/async-antipatterns>

Async C# Yet More on The Right Way

- Asynchronous Programming
<https://github.com/davidfowl/AspNetCoreDiagnosticScenarios/blob/master/AsyncGuidance.md#asynchronous-programming>
- Understanding Async, Avoiding Deadlocks in C#
<https://medium.com/rubrikkgroup/understanding-async-avoiding-deadlocks-e41f8f2c6f5d>
- The danger of async/await and .Result in one picture
<https://tooslowexception.com/the-danger-of-asyncawait-and-result-in-one-picture/>
- ConfigureAwait FAQ
<https://devblogs.microsoft.com/dotnet/configureawait-faq/>

Obrigado!

pricefy

DISTRIITO

NÚCLEO
DE VAREJO
Retail Lab

ESPM

INOVATIVA
BRASIL



TOP10
RANKING
RETAILTECHS

 medium.com/pricefy-labs

 github.com/leandrosilva

 leandrosilva.com.br