C# JEDI

C#: O Lado Brilhante da Programação

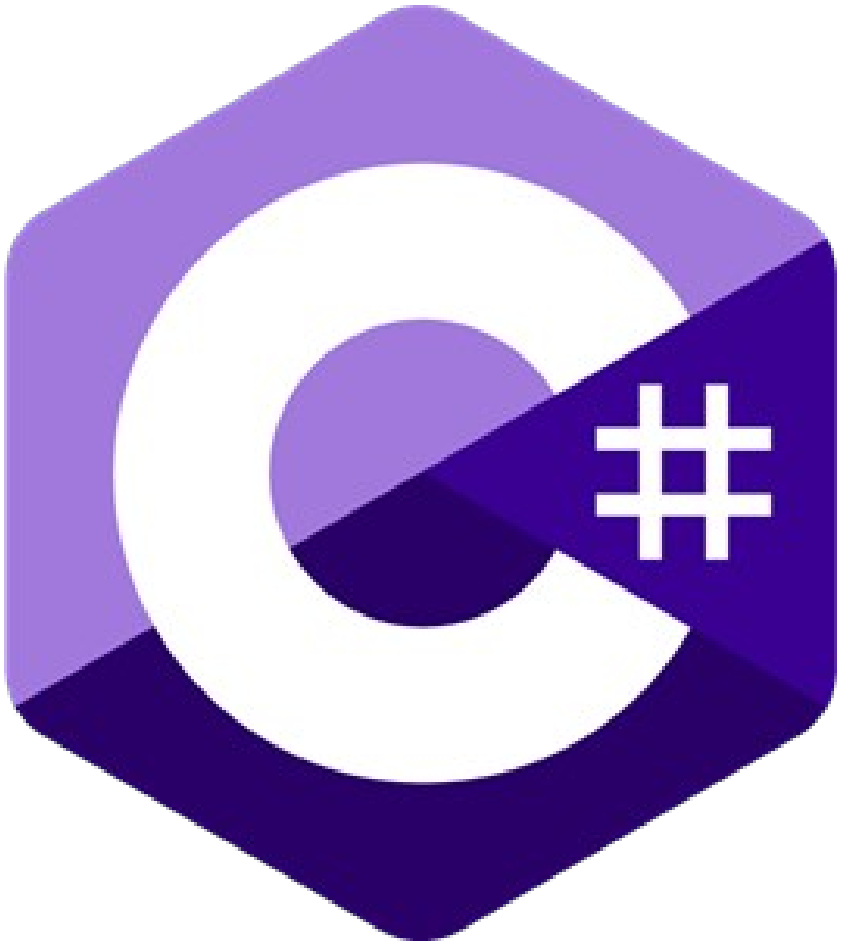
Explorando o Poder do *Async*/*Await*: Domine a Programação Assíncrona em C#

BRUNO HENRIQUE

PROGRAMAÇÃO ASSÍNCRONA EM C#

Simplificando o conceito

A programação assíncrona é uma técnica poderosa para melhorar a eficiência e a responsividade de aplicações, especialmente quando se trabalha com operações de I/O, como acesso a banco de dados ou chamadas de rede. Em C#, os principais recursos para implementar essa técnica são as palavras-chave *async* e *await*.



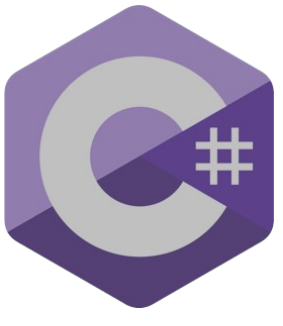
01

O que é Programação Assíncrona?

Neste capítulo, vamos explorar o conceito de programação assíncrona e como ele se diferencia da programação síncrona. Você entenderá a importância de operações não bloqueantes e como elas podem melhorar a experiência do usuário e o desempenho da aplicação.

O que é Programação Assíncrona?

A programação assíncrona permite que um programa execute operações longas, como leitura de arquivos ou chamadas de *APIs*, sem bloquear a execução do restante do código. Isso é crucial para manter a interface de usuário responsiva ou para aumentar o *throughput* de um servidor.



02

Async e Await: A Dupla Dinâmica

Aqui, apresentaremos as palavras-chave *async* e *await*, que são essenciais para a implementação de métodos assíncronos em C#. Explicaremos como utilizá-las para executar operações longas sem bloquear a execução do restante do código, com exemplos práticos para ilustrar seu uso.

Async e Await: A Dupla Dinâmica

Conceito Básico

As palavras-chave *async* e *await* são usadas para definir métodos assíncronos em C#. Um método assíncrono pode executar uma operação longa e retornar ao chamador antes que a operação termine, permitindo que outras tarefas sejam realizadas nesse meio tempo.

Exemplo:

Busca de Dados de uma API

```
1  public async Task<string> FetchDataFromApiAsync(string url)
2  {
3      using (HttpClient client = new HttpClient())
4      {
5          HttpResponseMessage response = await client.GetAsync(url);
6          response.EnsureSuccessStatusCode();
7          string responseBody = await response.Content.ReadAsStringAsync();
8          return responseBody;
9      }
10 }
```

Neste exemplo, a função *FetchDataFromApiAsync* é assíncrona e faz uma chamada a uma API. O *await* é usado para aguardar o resultado das chamadas *GetAsync* e *ReadAsStringAsync* sem bloquear o restante da aplicação.

03

Evitando Deadlocks com ConfigureAwait

Neste capítulo, abordaremos um problema comum em programação assíncrona: os *deadlocks*. Discutiremos como o uso inadequado de contextos de sincronização pode levar a situações de *deadlock* e como o *ConfigureAwait(false)* pode ser usado para evitá-los, garantindo uma execução fluida do seu código.

Evitando Deadlocks com ConfigureAwait

O Problema do Deadlock

Deadlocks podem ocorrer em aplicações de interface gráfica quando o contexto de sincronização não é liberado. Para evitar isso, use *ConfigureAwait(false)* em bibliotecas ou camadas não relacionadas à UI.

Exemplo

Uso de ConfigureAwait

```
1  public async Task<string> FetchDataWithNoDeadlockAsync(string url)
2  {
3      using (HttpClient client = new HttpClient())
4      {
5          HttpResponseMessage response = await client.GetAsync(url).ConfigureAwait(false);
6          response.EnsureSuccessStatusCode();
7          string responseBody = await response.Content.ReadAsStringAsync().ConfigureAwait(false);
8          return responseBody;
9      }
10 }
```

Nesse exemplo, *ConfigureAwait(false)* evita que o código tente voltar ao contexto de sincronização original, prevenindo *deadlocks*.

04

Lidando com Exceções em Métodos Assíncronos

Lidar com exceções em métodos assíncronos pode ser desafiador. Neste capítulo, exploraremos como capturar e tratar exceções de maneira eficaz em métodos assíncronos, garantindo que seu código seja robusto e seguro contra falhas inesperadas.

Lidando com Exceções em Métodos Assíncronos

Captura de Exceções

Exceções em métodos assíncronos podem ser capturadas usando blocos *try-catch*, assim como em métodos síncronos. No entanto, o tratamento pode exigir uma compreensão de como as tarefas são gerenciadas.

Exemplo

Tratamento de Exceções

```
1  public async Task<string> FetchDataSafelyAsync(string url)
2  {
3      try
4      {
5          using (HttpClient client = new HttpClient())
6          {
7              HttpResponseMessage response = await client.GetAsync(url);
8              response.EnsureSuccessStatusCode();
9              return await response.Content.ReadAsStringAsync();
10         }
11     }
12     catch (HttpRequestException e)
13     {
14         // Log e.Exception.Message
15         return "Error fetching data.";
16     }
17 }
```

Aqui, se uma exceção como *HttpRequestException* for lançada, ela será capturada e uma mensagem de erro será retornada.

05

Paralelismo com Task.WhenAll

Para maximizar a eficiência, é essencial entender como executar múltiplas tarefas simultaneamente. Neste capítulo, veremos como usar *Task.WhenAll* para rodar tarefas em paralelo, otimizando o tempo de execução e aproveitando ao máximo os recursos do sistema.

Paralelismo com Task.WhenAll

Executando Tarefas em Paralelo

O método *Task.WhenAll* permite executar várias tarefas assíncronas em paralelo, aumentando a eficiência da aplicação.

Exemplo

Consultando Múltiplas APIs

```
1  public async Task FetchMultipleApisAsync()
2  {
3      var url1 = "https://api.example.com/data1";
4      var url2 = "https://api.example.com/data2";
5
6      var task1 = FetchDataFromApiAsync(url1);
7      var task2 = FetchDataFromApiAsync(url2);
8
9      await Task.WhenAll(task1, task2);
10
11     var data1 = task1.Result;
12     var data2 = task2.Result;
13
14     // Processar os dados...
15 }
```

Nesse caso, *Task.WhenAll* é usado para esperar a conclusão de várias tarefas, permitindo que elas sejam executadas simultaneamente.

Conclusão

Conclusão

A programação assíncrona em C# é uma ferramenta essencial para criar aplicações mais eficientes e responsivas. Usando *async* e *await*, você pode melhorar a performance do seu código, evitando bloqueios e aproveitando melhor os recursos do sistema. Com esses conceitos e exemplos, você está pronto para explorar ainda mais as capacidades do C# em ambientes assíncronos.

AGRADECIMENTOS

OBRIGADO POR LER ATÉ AQUI

Esse Ebook foi gerado por IA, e diagramado por humano.
O passo a passo se encontra no meu Github

.

Esse conteúdo foi gerado com fins didáticos de construção, não foi realizado uma validação cuidadosa humana no conteúdo e pode conter erros gerados por uma IA.

https://github.com/BrunoHCS/prompts-create-a-ebook