Concurrencia

Contenido

[Concurrencia 2](#_Toc180228502)

[1. Programación Asíncrona 2](#_Toc180228503)

[1.1. Múltiples Tareas Task.WhenAll 2](#_Toc180228504)

[1.2. Nuevas Tareas Task.Run 3](#_Toc180228505)

[1.3. Limitando Tareas SemaphoreSlim 4](#_Toc180228506)

[1.4. IProgress 5](#_Toc180228507)

[1.5. Task.WhenAny 5](#_Toc180228508)

[1.6. Token Cancellation 7](#_Toc180228509)

[1.7. Cancelando bucles 7](#_Toc180228510)

[1.8. Cancelando por TimeOut 8](#_Toc180228511)

[1.9. Respuestas asincrónicas y procesamiento síncrono 8](#_Toc180228512)

[1.10. Contexto de sincronización 9](#_Toc180228513)

[1.11. Patrón Reintento 11](#_Toc180228514)

[1.12. Patrón Una sola tarea 11](#_Toc180228515)

[1.13. Controlando el resultado de la tarea. TaskCompletionSource 13](#_Toc180228516)

[1.14. Cancelando Tareas 13](#_Toc180228517)

[1.15. ValueTask 15](#_Toc180228518)

[2. Streams Asíncronos 16](#_Toc180228519)

[2.1. IEnumerable y Yield 16](#_Toc180228520)

[2.2. Streams asíncronos (IAsyncEnumerable) y Yield 17](#_Toc180228521)

[3. Antipatrones programación Asíncrona 18](#_Toc180228522)

[3.1. Síncrono dentro de un Asíncrono – Bloqueo mutuo (deadlock) sync-over-async 19](#_Toc180228523)

[3.2. Asíncrono dentro de un síncrono 19](#_Toc180228524)

[3.3. Async void – Altamente peligroso 20](#_Toc180228525)

[3.4. Evitar Task.Factory.StarNew – preferer Task.Run 21](#_Toc180228526)

[3.5. Hacer Dispose a los CancellationToken - Timers 21](#_Toc180228527)

[3.6. Hacer Dispose a los Streams 22](#_Toc180228528)

[3.7. Paralelismo 23](#_Toc180228529)

[3.8. Parallel.For 23](#_Toc180228530)

[3.9. Task.WhenAll vs Parallel.For 25](#_Toc180228531)

[3.10. Parallel.ForEach 25](#_Toc180228532)

[3.11. Observaciones Parallel 27](#_Toc180228533)

[3.12. Métodos atómicos 27](#_Toc180228534)

[3.13. Seguridad en hilos (thread – safe) 27](#_Toc180228535)

[3.14. Race condition (condición de carrera) 28](#_Toc180228536)

[3.15. Interlocked – Operaciones simples atómicas 28](#_Toc180228537)

[3.16. Lock 28](#_Toc180228538)

[3.17. ThreadStatic 29](#_Toc180228539)

[3.18. Parallel LINQ (PLINQ) 29](#_Toc180228540)

# Concurrencia

# Programación Asíncrona

# Múltiples Tareas Task.WhenAll

Vid 18

Pasar múltiples tareas a procesamientos y notifica cuando todas terminan.

[ThreadStatic]: me permite tener una copia particular por hilo de una variable

//0000000000000001

//0000000000000002

creditCards.Add(i.ToString().PadLeft(16, '0'));

var stopWatch = new Stopwatch();

stopWatch.Start();

MessageBox.Show($"Operation finalized in {stopWatch.ElapsedMilliseconds / 1000.0} segundos)}")

private async Task ProcessCards(List<string> cards)

{

var tasks = new List<Task>();

foreach (var card in cards)

{

var json = JsonConvert.SerializeObject(card);

var content = new StringContent(json, Encoding.UTF8, "application/json");

var answerTask = httpClient.PostAsync($"{apiURL}/creditcards", content);

tasks.Add(answerTask);

}

await Task.WhenAll(tasks);

}

# Nuevas Tareas Task.Run

Vid 19

Esto esta corriendo en el hilo UI

private async Task ProcessCards(List<string> cards)

{

var tasks = new List<Task>();

foreach (var card in cards)

{

var json = JsonConvert.SerializeObject(card);

var content = new StringContent(json, Encoding.UTF8, "application/json");

var answerTask = httpClient.PostAsync($"{apiURL}/creditcards", content);

tasks.Add(answerTask);

}

await Task.WhenAll(tasks);

}

Por eso se demora en iniciar el loading, pq el hilo principal se desbloquea cuando llega a await Task.WhenAll(tasks);

* Action se llama =>

Task.Run(() =>

//Liberamos el hilo UI

await Task.Run(() =>

{

};

var cards = await GetCreditCardsListAsync(1000);

var stopWatch = new Stopwatch();

stopWatch.Start();

try

{

//await ProcessCards(cards);

await ProcessCardsRunAsync(cards);

}

private async Task ProcessCardsRunAsync(List<string> cards)

{

var tasks = new List<Task>();

//Liberamos el hilo UI

await Task.Run(() =>

{

foreach (var card in cards)

{

var json = JsonConvert.SerializeObject(card);

var content = new StringContent(json, Encoding.UTF8, "application/json");

var answerTask = httpClient.PostAsync($"{apiURL}/creditcards", content);

tasks.Add(answerTask);

}

});

await Task.WhenAll(tasks);

}

# Limitando Tareas SemaphoreSlim

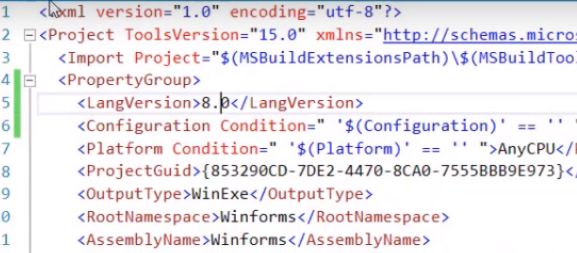
Vid 20

Throttling

Cuando tenemos mucha data y abrumamos otra parte del software con el procesamiento de la misma.

Se necesita C Sharp 8

Para cambiar la versión de C, propiedades del proyecto, unload Project, y pegar el tag <LangVersion>8.0</LangVersion>



# IProgress

Vid 22

Disparar un evento cada vez que ocurre algo, se puede utilizar para notificar el progreso de un conjunto de tareas.

var reportProgress = new Progress<int>(ReportProgressCards);

private async Task ProcessCardsRunProgressAsync(List<string> cards, IProgress<int> progress = null)

{

var tasks = new List<Task>();

int index = 0;

//Liberamos el hilo UI

await Task.Run(() =>

{

foreach (var card in cards)

{

var json = JsonConvert.SerializeObject(card);

var content = new StringContent(json, Encoding.UTF8, "application/json");

var answerTask = httpClient.PostAsync($"{apiURL}/creditcards", content);

if (progress != null)

{

index++;

var percentage = (double)index / cards.Count;

var percentageRounded = (int)Math.Round(percentage \* 100, 0);

//Report se encarga de ejecutar el método ReportProgressCards

progress.Report(percentageRounded);

}

tasks.Add(answerTask);

}

});

await Task.WhenAll(tasks);

}

private void ReportProgressCards(int percentage)

{

pgProcess.Value = percentage;

}

# Task.WhenAny

Vid 23

Cuando no es necesario reportar el progreso en un bucle, sin no que se necesita cuando una tarea termina, cualquiera. Podemos saber en tiempo real cuales tareas han sido completadas.

Task.IsFaulted: tarea terminada con error

Task.IsComplete: tarea terminada con error

Task.IsCancel: tarea terminada con error

//Ya no vamos a esperar a que finalicen todas las tareas

//var answers = await Task.WhenAll(tasks);

var answersTasks = Task.WhenAll(tasks);

if (progress != null)

{

while (await Task.WhenAny(answersTasks, Task.Delay(1000)) != answersTasks)

{

//el != compara si ya se terminaron las tareas (answerTask) y finaliza el while

//el WhenAny compara cual tarea está mas demorada y devuelve la de menos tiempo y sigue ejecutandose el While

var taskCompleted = tasks.Where(task => task.IsCompleted).Count();

var percentage = (double)taskCompleted / cards.Count;

var percentageRounded = (int)Math.Round(percentage \* 100, 0);

//Report se encarga de ejecutar el método ReportProgressCards

progress.Report(percentageRounded);

}

}

//Si ya las tareas fueron completadas, no significa que con el await de abajo

//se va a volver a realizar la tarea. Esperar una tarea dos veces no significa

//que se va a ejecutar dos veces

var answers = await answersTasks;

var cardsRejected = new List<string>();

foreach (var answer in answers)

{

var content = await answer.Content.ReadAsStringAsync();

var answerCard = JsonConvert.DeserializeObject<AnswerCard>(content);

if (!answerCard.IsApproved)

{

cardsRejected.Add(answerCard.Card);

}

}

Console.WriteLine("Cards rejected:");

foreach (var card in cardsRejected)

{

Console.WriteLine(card);

}

# Token Cancellation

Vid 24

Cuando hay tareas largas es bueno poder permitir cancelar las tareas.

try

{

await ProcessCardsWhenAnyAsync(cards, reportProgress, cancellationTokenSource.Token);

}

catch (HttpRequestException ex)

{

MessageBox.Show(ex.Message);

}

catch (TaskCanceledException ex)

{

MessageBox.Show($"The operation was canceled. {ex.Message }");

}

private async Task ProcessCardsWhenAnyAsync(List<string> cards

, IProgress<int> progress = null,

CancellationToken cancellationToken = default)

{

//El default sirve para que sea una variable opcional

private async Task ProcessCardsWhenAnyAsync(List<string> cards

, IProgress<int> progress = null,

CancellationToken cancellationToken = default)

{

…

try

{

var taskInternal = await httpClient.PostAsync($"{apiURL}/creditcards", content, cancellationToken);

private void btnCancel\_Click(object sender, EventArgs e)

{

//el ? sirve para validar que no sea nulo

cancellationTokenSource?.Cancel();

}

Cuando se cancela una tarea, se origina una excepción

# Cancelando bucles

Vid 25

Cuando hay tareas bucles muy largos y necesito poder cancelarlos.

var cards = await GetCreditCardsCancellationAsync(100, cancellationTokenSource.Token);

private async Task<List<string>> GetCreditCardsCancellationAsync(int quantityCards

, CancellationToken cancellationToken = default)

{

return await Task.Run(async () =>

{

var creditCards = new List<string>();

for (int i = 0; i < quantityCards; i++)

{

//creditCards.Add(i.ToString().PadLeft(16, '0'));

//Vid 25 cancelando bucles

await Task.Delay(1000);

creditCards.Add(i.ToString().PadLeft(16, '0'));

Console.WriteLine($"Han sido generadas {creditCards.Count} tarjetas");

if (cancellationToken.IsCancellationRequested)

{

//Me dice si el generador de tokens ha solicitado la cancelación del token

//break;

////or

throw new TaskCanceledException();

}

}

return creditCards;

# Cancelando por TimeOut

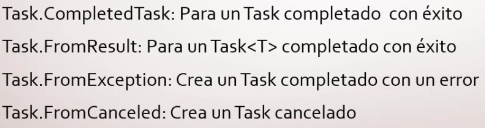
Vid 26

cancellationTokenSource = new CancellationTokenSource();

cancellationTokenSource.CancelAfter(TimeSpan.FromSeconds(3));

# Respuestas asincrónicas y procesamiento síncrono

Vid 27



Cuando yo necesito que se ejecuten tareas, pero no necesito hacer nada con ellas todavía.

private Task<List<string>> GetCreditCardsMock(int quantityCards

, CancellationToken cancellationToken = default)

{

var cards = new List<string>();

cards.Add("0000000001");

return Task.FromResult(cards);

}

private Task GetTaskErrorMock(int quantityCards

, CancellationToken cancellationToken = default)

{

return Task.FromException(new ApplicationException());

}

private Task GetTaskCancelledMock(int quantityCards

, CancellationToken cancellationToken = default)

{

cancellationTokenSource = new CancellationTokenSource();

return Task.FromCanceled(cancellationTokenSource.Token);

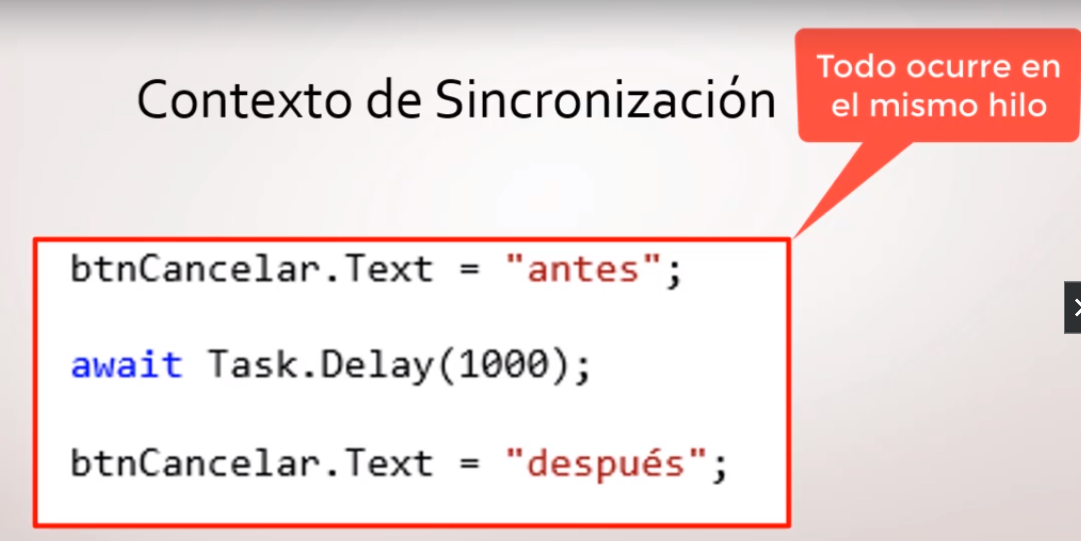
}

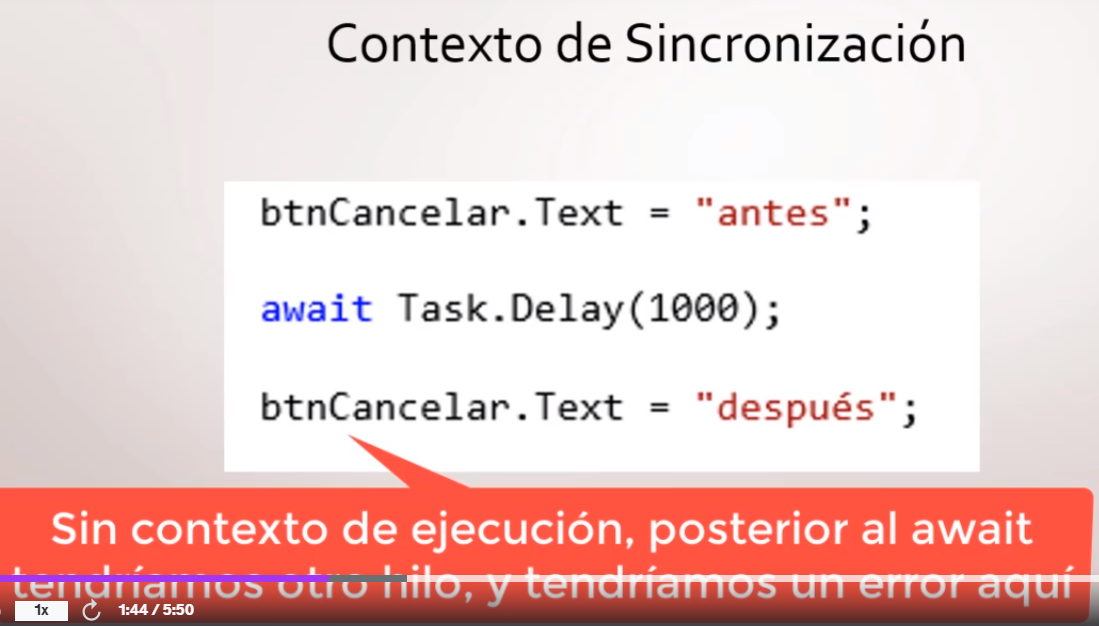
# Contexto de sincronización

Vid 28

SynchronizationContext

No todos los hilos son iguales y pueden hacer lo mismo. Ejemplo: el hilo UI de aplicaciones de escritorio (WinForms) o WPF. Y queremos que la tarea que termina, se resuma la ejecución en el mismo hilo principal.





NET Core es un framework sin contexto de sincronización.

Console.WriteLine($"Thread before await: {Thread.CurrentThread.ManagedThreadId}");

Que no haya contexto de sincronización implica que es posible que el mismo hilo que inicia una petición HTTP no sea el mismo hilo que termine dicha petición.

Cuando no hay UI no es relativamente importante.

Es posible que no tener contexto de sincronización haga que se ejecute mas rápido, al poder utilizar cualquier hilo.

En WinForms y WPF el hilo principal se suspende mientras realiza las actividades. Componentes creados en la UI deben ser manejados por el hilo UI.

**ConfigureAwait(false)** hace que no se necesite esperar o bloquear al mismo hilo que inicio la petición

ConfigureAwait(true); hace que se ejecute en el mismo contexto.

await Task.Delay(500).ConfigureAwait(continueOnCapturedContext: false);

CheckForIllegalCrossThreadCalls = true;

CheckForIllegalCrossThreadCalls = true; evalúa si se van a ejecutar llamadas ilegales entre hilos.

ConfigureAwait(false), es bueno o útil, utilizarlo cuando vamos a ir a proceso que va a ejecutar cosas asíncronas y no necesitamos que se ejecute en el mismo hilo, y queremos mejorar el rendimiento. No aplica para NET Core por lo que no hay contexto de sincronización en NET Core, en NET Core no hay hilos especiales.

# Patrón Reintento

Vid 30

SynchronizationContext

private async Task Retry(Func<Task> f, int retries = 3, int waitTime = 500)

{

for (int i = 0; i < retries; i++)

{

try

{

await f();

break;//para salir del for, si la operation is successful

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

await Task.Delay(waitTime);

}

}

}

Funciones anónimas: async () =>

# Patrón Una sola tarea

Vid 31

/// <summary>

/// Only one taks, the first in end to complete

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private async void btnOneTask\_Click(object sender, EventArgs e)

{

loadingGif.Visible = true;

cancellationTokenSource = new CancellationTokenSource();

var token = cancellationTokenSource.Token;

var names = new string[] { "Dani", "Rocko", "Camila", "Juan" };

///Comentamos esto y lo vamos a meter en un método general

//var taskHTTP = names.Select(x => GetGreetingsDelayCancel(x, token));

//var task = await Task.WhenAny(taskHTTP); //cualquiera de las tareas que termine

//var content = await task;

//Console.WriteLine(content.ToUpper());

//cancellationTokenSource.Cancel();

///

//var taskHTTP = names.Select(x =>

//{

// Func<CancellationToken, Task<string>> function = (cancelT) => GetGreetingsDelayCancel(x, cancelT);

// return function;

//});

//var content = await ExecuteOneTask(taskHTTP);

//Console.WriteLine(content.ToUpper());

///Ajustamos el llamado anterior y creamos un nuevo método para tener mas llamadas landa

///para llamar distintas o varias funciones. Solo se va a ejecutar una sola tarea, una sola función

var content = await ExecuteOneTask(

(ct) => GetGreetingsDelayCancel("Dani", ct),

(ct) => GetGoodbyeDelayCancel("Dani", ct)

);

Console.WriteLine(content.ToUpper());

loadingGif.Visible = false;

}

var content = await ExecuteOneTask(

(ct) => GetGreetingsDelayCancel("Dani", ct),

(ct) => GetGoodbyeDelayCancel("Dani", ct)

);

private async Task<T> ExecuteOneTask<T>(IEnumerable<Func<CancellationToken, Task<T>>> functions)

{

var cts = new CancellationTokenSource();

var tasks = functions.Select(funcion => funcion(cts.Token));

var task = await Task.WhenAny(tasks);

cts.Cancel();

return await task;

}

private async Task<T> ExecuteOneTask<T>(params Func<CancellationToken, Task<T>>[] functions)

{

var cts = new CancellationTokenSource();

var tasks = functions.Select(funcion => funcion(cts.Token));

var task = await Task.WhenAny(tasks);

cts.Cancel();

return await task;

}

# Controlando el resultado de la tarea. TaskCompletionSource

Vid 32

Con TaskCompletionSource prodemos crear una tarea y controlar su status.

private async void btnStartStatusControlled\_Click(object sender, EventArgs e)

{

loadingGif.Visible = true;

var task = EvaluateValue(txtInputStatusValue.Text);

Console.WriteLine("Begin");

Console.WriteLine($"Is Completed {task.IsCompleted}");

Console.WriteLine($"Is Canceled {task.IsCanceled}");

Console.WriteLine($"Is Faulted {task.IsFaulted}");

try

{

await task;

}

catch (Exception ex)

{

Console.WriteLine($"Exception: {ex.Message}");

}

Console.WriteLine("End");

Console.WriteLine("");

loadingGif.Visible = false;

}

public Task EvaluateValue(string value)

{

var tcs = new TaskCompletionSource<object>(TaskCreationOptions.RunContinuationsAsynchronously);

if (value == "1")

{

tcs.SetResult(null);

}

else if (value == "2")

{

tcs.SetCanceled();

}

else

{

tcs.SetException(new ApplicationException($"Invalid value {value}"));

}

return tcs.Task;

}

# Cancelando Tareas

Vid 33

Cancelar tareas que no tienen token de cancelación.

cancellationToken.Register Me permite tener un delegado, es decir, una función que va a ejecutarse cuando el token sea cancelado

public static class TaskExtensionMethods

{

public static async Task<T> WithCancellation<T>(this Task<T> task, CancellationToken cancellationToken)

{

var tcs = new TaskCompletionSource<object>(TaskCreationOptions.RunContinuationsAsynchronously);

//cancellationToken.Register Me permite tener un delegado, es decir, una función que va a

//ejecutarse cuando el token sea cancelado

using (cancellationToken.Register(state =>

{

((TaskCompletionSource<object>)state).TrySetResult(null);

}, tcs))

{

var taskResult = await Task.WhenAny(task, tcs.Task);

if (taskResult == tcs.Task) //Significa que el usuario canceló la tarea

{

throw new OperationCanceledException(cancellationToken);

}

return await task;

}

}

}

loadingGif.Visible = true;

cancellationTokenSource = new CancellationTokenSource();

try

{

var result = await Task.Run(async () =>

{

await Task.Delay(5000);

return 7;

}).WithCancellation(cancellationTokenSource.Token);

Console.WriteLine($"{result.ToString()}");

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

finally { cancellationTokenSource.Dispose(); }

loadingGif.Visible = false;

# ValueTask

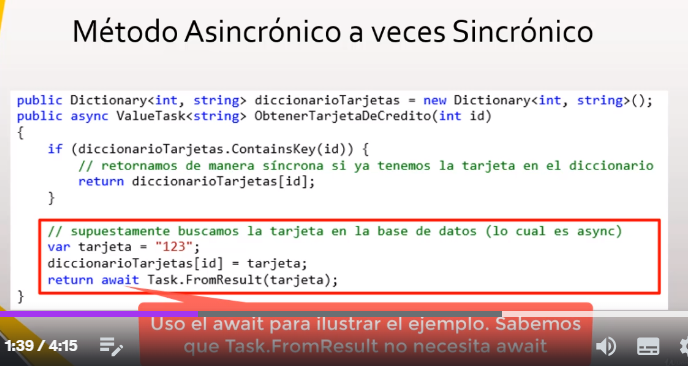
Vid 34

La idea de ValueTask es performance en donde haya escenario se alta demanda.

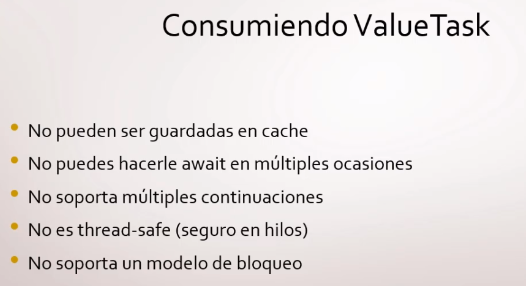
ValueTask es un struct (es un tipo de valor y no un tipo de referencia como Task).

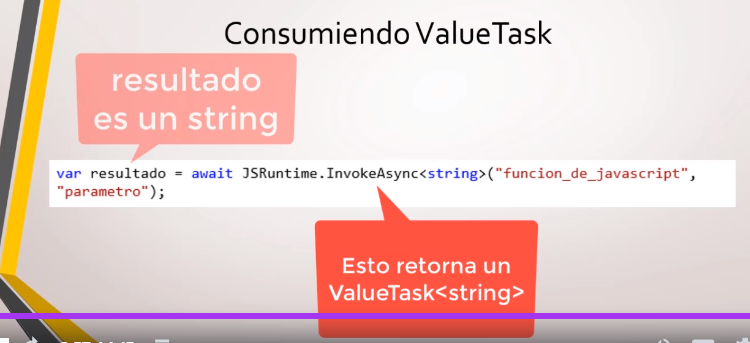
Dos escenarios para poderlo utilizar:

* Método asíncrono casi siempre síncrono.
* Método usado con mucha frecuencia y el costo sea importante.



Performance: solo usamos ValueTask si el análisis de performance lo justifica.





# Streams Asíncronos

Vid 36

# IEnumerable y Yield

Vid 37

**IEnumerable**: Interfaz que nos permite realizar iteraciones sobre un tipo. Una lista implementa la interfaz IEnumerable.

**Yield**: es la generación de uno en uno de un iterable.

**Stream**: secuencia de datos

private IEnumerable<string> GenerateNames()

{

yield return "Dani";

yield return "Camilo";

}

private void btnIEnumerable\_Click(object sender, EventArgs e)

{

loadingGif.Visible = false;

foreach (var name in GenerateNames())

{

Console.WriteLine(name);

}

loadingGif.Visible = false;

}

El yield va retornando uno a uno, según la iteración del foreach. La ejecución del método se detiene luego de enviar cada valor y cada que vuelve, el método recuerda en cual return o valor va, y lo devuelve.

# Streams asíncronos (IAsyncEnumerable) y Yield

Vid 38

Probar si da con IAsyncEnumerable, si no, instalar paquete nuget



**Escenarios**:

Consumo de un servicio que tiene páginación, no va a retornar todos los valores de inmediato si no por partes.

En el código hay otro ejemplo antes de utilizar el EnumeratorCancellation

**EnumeratorCancellation**: atributo especial para trabajar con el WithCancellation y que reconozca que es un método que recibe datos (parámetros) con cancelación de token.

private async Task ProcessNames(IAsyncEnumerable<string> names)

{

try

{

await foreach (var name in names.WithCancellation(cancellationTokenSource.Token))

{

Console.WriteLine(name);

}

}

catch (TaskCanceledException cex)

{

Console.WriteLine("Operation cancelled");

}

finally

{

cancellationTokenSource?.Dispose();

}

}

private async IAsyncEnumerable<string> GenerateNamesAsync2([EnumeratorCancellation] CancellationToken token = default)

{

yield return "Dani";

await Task.Delay(500, token);

yield return "Camilo 0.5 s";

await Task.Delay(2000, token);

yield return "Camilo 2 s";

await Task.Delay(500, token);

yield return "Camilo 0.5";

await Task.Delay(300, token);

yield return "Camilo 0.3";

}

/// <summary>

/// esto es para cuando no tengo control del método para cancelar, si no que tengo es control sobre el retorno del método

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private async void button3\_Click(object sender, EventArgs e)

{

loadingGif.Visible = false;

cancellationTokenSource = new CancellationTokenSource();

var namesEnumerable = GenerateNamesAsync2();

await ProcessNames(namesEnumerable);

Console.WriteLine("Finish");

loadingGif.Visible = false;

}

# Antipatrones programación Asíncrona

Vid 41

# Síncrono dentro de un Asíncrono – Bloqueo mutuo (deadlock) sync-over-async

Vid 42

Result bloquea el hilo

Configure await lo que hace es no utilizar el hilo principal

//Antipatrones

private async Task<string> GetValue()

{

await Task.Delay(1000);

////2da Solución, con esto no se bloquea el hilo principal, animaciones y demás,

////pero al menos no se crashea la app

//await Task.Delay(1000).ConfigureAwait(false);

return "Dani";

}

private void button7\_Click(object sender, EventArgs e)

{

loadingGif.Visible = true;

var value = GetValue().Result;

Console.WriteLine(value);

////1ra solución, no bloquar el hilo principal

//var value = GetValue();

//Console.WriteLine(value);

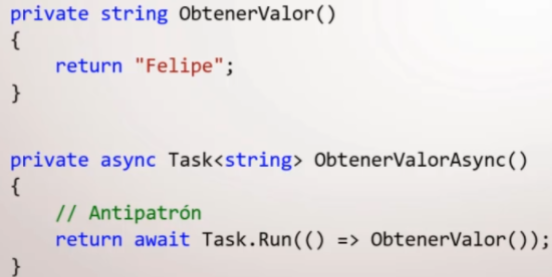
loadingGif.Visible = false;

}

# Asíncrono dentro de un síncrono

Vid 43

Cuando utilizamos programación asíncrona, buscamos escalabilidad, que no se congele la aplicación.



//Asincrono dentro de un síncrono

private string GetValueSync()

{

return "Dani";

}

private async Task<string> GetValueAsync()

{

//Antipatrón

return await Task.Run(() => GetValueSync());

}

Con esto estamos volviendo el método asíncrono, en síncrono.

El consumo de hilos tiene su costo, no todo debe ser asíncrono, debe valer la pena.

# Async void – Altamente peligroso

Vid 44

Si un async void lanza una excepción, toda la app se va a detener.

Un método asíncrono que retorna un Task y lanza una excepción, dispara un evento, el cual permite que controlemos la excepción y evitar que escale y tumbe la aplicación. Con un async void, este mecanismo no funciona.

Excepción: event handlers

**Si tenemos una excepción en un método ASYNC VOID todo colapsa.**

/// <summary>

/// Antipatrón

/// </summary>

private async void OperationVoidAsync()

{

await Task.Delay(1);

throw new ApplicationException();

}

Tener un try catch no soluciona el problema tampoco.

Así, la exception queda dentro del hilo del Task

private async Task OperationTaskAsync()

{

await Task.Delay(1);

throw new ApplicationException();

}

Si temenos un void con excepción, solo colapsa el método, pero no toda la app o api

# Evitar Task.Factory.StarNew – preferer Task.Run

Vid 45

Task.Run

//var resultStartNew = await await Task.Factory.StartNew(async () =>

var resultStartNew = await Task.Factory.StartNew(async () =>

{

//Este delegado queda envuelto en otro Task. StarNew quedaria

//Task<Task<int>>, hay que colcoar otro await

// o utilizar unwrap

await Task.Delay(1000);

return 7;

}).Unwrap();

# Hacer Dispose a los CancellationToken - Timers

Vid 46

A los timers debemos hacerles siempre dispose. Los cancellationToken internamente utilizan timers.

Por regla general, si algo hereda de dispose, deberíamos hacerles siempre dispose.

3 formas:

Var tiempoLimitie = TimeSpan.FromSeconds(5);

Try

{

cancellationTokenSource = new CancellationTokenSource(tiempoLimite);

}

Finally{

cancellationTokenSource.Dispose();

}

* Hacer uso de Using

Using(var cts2 = new CancellationTokenSource(tiempoLimite)){

}

* Hacer uso de using vble

Using var cts3 = new CancellationTokenSource(tiempoLimite);

# Hacer Dispose a los Streams

Vid 47

Cuando usamos Streams es posible que existan datos en buffer. Cuando se hace el dispose de manera síncrona del stream, estos datos del buffer se van a transferir de manera síncrona. Esto puede significar el antipatrón síncrono dentro de asíncrono.

private async Task MethodAcyncEx()

{

using (var streamWriter = new StreamWriter(stream))

{

await streamWriter.WriteAsync("Hello world");

}

//es posible que cuando se haga el dispose aún existan datos en el buffer,

//estos datos se van a transferir de manera sincrona

//solution 1

await using (var streamWriter = new StreamWriter(stream))

{

await streamWriter.WriteAsync("Hello world");

}

//solution 2

using (var streamWriter = new StreamWriter(stream))

{

await streamWriter.WriteAsync("Hello world");

await streamWriter.FlushAsync();

}

}

**Flush** es la operación que hace el vaciado de toda la información que hay en el buffer hacia su destino.

Lo mejor es usar Flush

# Paralelismo

Vid 47

Paralelismo nos sirve para dividir una tarea en distintas partes y trabajar esas partes de manera distinta, usando recursos del computador (multihilos – procesadores multinucleos0)

No es bueno para ASP.NET y ASP.NET Core

Operaciones de CPU: procesos aritméticos, trabajar con imágenes, operaciones que dependen de nosotros y no externas a la aplicación.

Se debe evaluar el costo de usarlo, no siempre es bueno usarlo pq es costoso y depende de la cantidad de trabajo.

stopwatch.Restart();

// Parte paralelo

var tareasEnumerable = images.Select(async imagen => await ProcesarImagen(destionationBaseParallel, imagen));

await Task.WhenAll(tareasEnumerable);

var tiempoEnParalelo = stopwatch.ElapsedMilliseconds / 1000.0;

Console.WriteLine("Paralelo - duración en segundos: {0}", tiempoEnParalelo);

# Parallel.For

Vid 53

Paralelismo nos sirve tareas que no sean IO.

private void button10\_Click(object sender, EventArgs e)

{

loadingGif.Visible = true;

Console.WriteLine("Secuential");

for (int i = 0; i < 11; i++)

{

Console.WriteLine(i);

}

Console.WriteLine("Parallel");

Parallel.For(0, 11, i => Console.WriteLine(i));

loadingGif.Visible = false;

}

Parallel.For – velocidad

public static void MultiplicarMatricesSecuencial(double[,] matA, double[,] matB,

double[,] result)

{

int matACols = matA.GetLength(1);

int matBCols = matB.GetLength(1);

int matARows = matA.GetLength(0);

for (int i = 0; i < matARows; i++)

{

for (int j = 0; j < matBCols; j++)

{

double temp = 0;

for (int k = 0; k < matACols; k++)

{

temp += matA[i, k] \* matB[k, j];

}

result[i, j] += temp;

}

}

}

public static void MultiplicarMatricesParalelo(double[,] matA, double[,] matB,

double[,] result,

CancellationToken token = default,

int maximoGradoParalelismo = -1)

{

int matACols = matA.GetLength(1);

int matBCols = matB.GetLength(1);

int matARows = matA.GetLength(0);

Parallel.For(0, matARows,

new ParallelOptions()

{

CancellationToken = token,

MaxDegreeOfParallelism = maximoGradoParalelismo

},

i =>

{

for (int j = 0; j < matBCols; j++)

{

double temp = 0;

for (int k = 0; k < matACols; k++)

{

temp += matA[i, k] \* matB[k, j];

}

result[i, j] += temp;

}

});

}

# Task.WhenAll vs Parallel.For

Vid 54

Si es I/O usar Task.WhenAll

Si es procesamiento CPU (operaciones aritméticas por ejemplo) usar Parallel.For

# Parallel.ForEach

Vid 55

public async Task ParallelForEach()

{

var directorioActual = AppDomain.CurrentDomain.BaseDirectory;

//need images in next folder, previous exercise

var carpetaOrigen = Path.Combine(directorioActual, @"D:\tmpBlazorFiles\secuentialResult");

var carpetaDestinoSecuencial = Path.Combine(directorioActual, @"D:\tmpBlazorFiles\foreach-secuencial");

var carpetaDestinoParalelo = Path.Combine(directorioActual, @"D:\tmpBlazorFiles\foreach-paralelo");

Utils.PrepararEjecucion(carpetaDestinoSecuencial, carpetaDestinoParalelo);

var archivos = Directory.EnumerateFiles(carpetaOrigen);

var stopwatch = new Stopwatch();

stopwatch.Start();

// Algoritmo secuencial

foreach (var archivo in archivos)

{

VoltearImagen(archivo, carpetaDestinoSecuencial);

}

var tiempoSecuencial = stopwatch.ElapsedMilliseconds / 1000.0;

Console.WriteLine("Secuencial - duración en segundos: {0}",

tiempoSecuencial);

stopwatch.Restart();

// Algoritmo en paralelo

Parallel.ForEach(archivos, archivo =>

{

VoltearImagen(archivo, carpetaDestinoParalelo);

});

var tiempoEnParalelo = stopwatch.ElapsedMilliseconds / 1000.0;

Console.WriteLine("Paralelo - duración en segundos: {0}",

tiempoEnParalelo);

Utils.EscribirComparacion(tiempoSecuencial, tiempoEnParalelo);

Console.WriteLine("fin");

}

private void VoltearImagen(string archivo, string carpetaDestino)

{

using (var image = new Bitmap(archivo))

{

image.RotateFlip(RotateFlipType.Rotate90FlipNone);

var nombreArchivo = Path.GetFileName(archivo);

var destino = Path.Combine(carpetaDestino, nombreArchivo);

image.Save(destino);

}

}

# Observaciones Parallel

Vid 56

Siempre realizar mediciones para saber si conviene o no utilizar paralelismo. El paralelismo tiene su costo. Todo depende de la cantidad de información a procesar.

# Métodos atómicos

Vid 60

Pueden ser utilizados en ambientes multihilos, pq garantizan determinismo (siempre obtendremos el mismo resultado sin importar cuantos hilos intenten acceder de manera simultánea).

Características:

* Si un hilo está ejecutando un método, otro hilo no puede ver un estado intermedio, es decir, no ha empezado o ya terminó, pero no en la mitad.
* La operación va a finalizar exitosamente o va a fallar sin realizar modificaciones.

Utilizar:

Locks: bloquear otros hilos

Colecciones concurrentes

# Seguridad en hilos (thread – safe)

Vid 61

Quiere decir que podemos ejecutar de manera simultanea un hilo sin causar ningún tipo de error.

Si en el método modificamos una **variable externa** podríamos tener problemas de resultados inesperados. Podemos usar **lock** para el acceso a dicha variable

Si usamos **objetos**, podemos usar **objetos inmutables**.

Funciones puras: son aquellas que retornan el mismo valor para los mismo argumentos y no causan efectos secundarios.

# Race condition (condición de carrera)

Vid 62

Cuando hay una variable compartida por varios hilos, y dichos hilos quieren modificar la variable de manera simultánea.

Se deben tener operaciones atómicas.

# Interlocked – Operaciones simples atómicas

Vid 63

Sirve para realizar operaciones básicas como incrementos, decrementos, sumar, etc.

# Lock

Vid 64

Se nuestro bloque de código se ejecuta por un hilo a la vez. Secuencial.

Lo que hagamos dentro de un lock debe ser relativamente rápido (por el bloqueo de hilos, esto afecta el performance de la aplicación).

private void Calllock()

{

loadingGif.Visible = true;

Console.WriteLine("Begin");

var valueIncremented = 0;

var valuePlus = 0;

////doesn´t work with Interlocked

//Parallel.For(0, 10000, number =>

//{

// Interlocked.Increment(ref valueIncremented);

////otro hilo puede volver a ejecutar la línea de arriba mientras se

////ejecuta la línea de abajo, por eso no sirve çon interlocked

// Interlocked.Add(ref valuePlus, valueIncremented);

//});

var mutex = new object();

Parallel.For(0, 10000, number =>

{

//inside of lock have only one thread

lock (mutex)

{

valueIncremented++;

valuePlus += valueIncremented;

}

});

Console.WriteLine($"value incremented: {valueIncremented}");

Console.WriteLine($"value total: {valuePlus}");

Console.WriteLine("---------");

Console.WriteLine("End");

loadingGif.Visible = false;

}

# ThreadStatic

Vid 65

Se utiliza para indicar que un campo estático es único para cada hilo.

Ramdom no es segura usando varios hilos

# Parallel LINQ (PLINQ)

Vid 66

LINQ pero en paralelismo. Se puede definir máximo grado de paralelismo y cancellation Token.

# Antipatrones Paralelismo

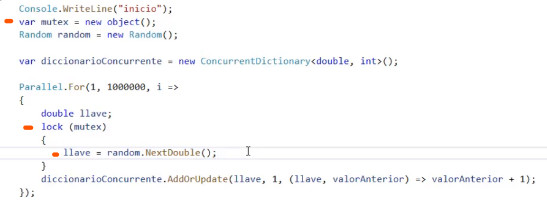
Vid 70

**Paralelismo innecesario**, cuando las operaciones son muy sencillas no conviene utilizar paralelismo por el consumo que esto requiere.

**Condiciones de carrera:** variables compartidas, se puede llegar a editar desde diferentes hilos la misma variable. Se deben manejar operaciones atómicas. Utilizar Interlock.Increment(ref valor); o locks.

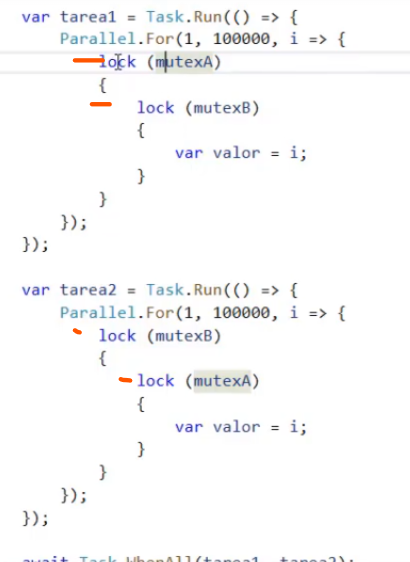
**Sobresaturación:** es cuando la cantidad de hilos creados por nuestra aplicación sobre pasa la capacidad de nuestro procesador. Evitar anidar ParallelFor dentro de ParallelFor

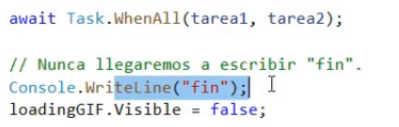
**Usando clases no seguras:** System.Random. Si se va a utilizar un Random, se debe usar un lock.



**Uso de locks** es peligroso si se hace locks de **varios objetos**. (dead lock). Tampoco usar **Lock(this)**.

Tampoco usar **lock(typeof(int))**{}





Nunca va a llegar a fin pq vamos a tener un dead lock. Se hace bloqueo de A para esperar a B y lo mismo en el siguiente, se bloqua A para esperar a B, tenemos un bloqueo mutuo.

d

Throttling