Asynchronous Programming Techniques & Best Practices

BY RICCARDO TERRELL - @RIKACE

Objectives

- Increase speed of code without unwanted side effects
  - Regardless the available cores
- Understand the implications of Asynchronous programming
- Improve scalability
- Best Practices for exploiting Asynchronous programming

Asynchronous programming is Functional

All asynchronous code is functional by nature.

"you have to turn your mind inside out."

That's regular-person-speak for "think functionally, not procedurally."

Asynchronous Workflows

- Software is often I/O-bound, it provides notable performance benefits
  - Connecting to the Database
  - Leveraging web services
  - Working with data on disks
- Network and disk speeds increasingly slower
- Not easy to predict when the operation will complete (non-deterministic)
- **IO bound functions can scale regardless of threads**
  - **IO bound computations can often “overlap”**
  - **This can even work a for huge numbers of computations**

Is it CPU-bound, or I/O-bound?

Asynchronous for I/O Bound

The CPU-bound computations receive input from the keyboard to do some work, and then print the result to the screen. In a single core machine, each computation mast be completed before proceed to the next one.

The I/O-bound computations are executed independently from the CPU, and the operation is done elsewhere. In this case several asynchronous database calls are executing simultaneously. Later a notification will inform the caller when the operation complete *(Callback)*.

ThreadPool

```
// table1.DataSource = LoadHousesSequentially(1,5);
// table1.DataBind();

public List<House> LoadHousesSequentially(int first, int last)
{
    var loadedHouses = new List<House>();

    for (int i = first; i <= last; i++) {
        House house = House.Deserialize(i);
        loadedHouses.Add(house);
    }

    return loadedHouses;
}
```

ThreadPool

request in
start1
end1
start2
end2
start3
end3
start4
end4
start5
end5
response out
500ms

ThreadPool

request in
Parallel.For
start1
end1
start5
end5
start2
end2
start3
end3
start4
end4
start1
1
end1
start2
2
end2
start4
4
end4
start5
5
end5
start3
3
end3
response out
~200ms

ThreadPool

request in
start1
start2
start3
start4
start5
end2
end5
end1
end3
end4
response out
~100ms

ThreadPool

```
// table1.DataSource = await LoadHousesAsync(1,5);
// table1.DataBind();

public async Task<List<House>> LoadHousesAsync(int first, int last)
{
    var tasks = new List<Task<House>>();

    for (int i = first; i <= last; i++)
    {
        Task<House> t = House.LoadFromDatabaseAsync(i);
        tasks.Add(t);
    }

    House[] loadedHouses = await Task.WhenAll(tasks);
    return loadedHouses.ToList();
}
```

Threadpool

Principles

- CPU-bound work means things like: LINQ-over-objects, or big iterations, or computational inner loops.
- Parallel.ForEach and Task.Run are a good way to put CPU-bound work onto the thread pool.
- Thread pool will gradually feel out how many threads are needed to make best progress.
- Use of threads will *never* increase throughput on a machine that's under load.

Guidance

- For IO-bound "work", use await rather than background threads.
- For CPU-bound work, consider using background threads via Parallel.ForEach or Task.Run, unless you're writing a library, or scalable server-side code.

Running I/O operation using a Task

```
Task.Run(()=>
{
    using(var stream = File.Open(filePath))
    {
        byte[] buffer = new byte[16384];
        var bytesRead = stream.Read(buffer, 0, buffer.Length);
    }
});
```

Async Programming motivation

OS threads are **expensive**, while lightweight threading alone is less interoperable.

Asynchronous programming **using callbacks is difficult**.

Unbounded parallelism – no hardware constraints

Asynchronous workflows

- Keep standard **programming model**
- Standard **exception handling** and **loops**
- **Sequential** and **parallel** composition

Parallel asynchronous programming

Parallel vs. asynchronous programming

- Two different problems
- C# **Tasks** are good for both
- F# **Async** mainly for asynchronous programming

Declarative parallelism

- Compose tasks to run in parallel

Task-based parallelism

- Create tasks and wait for them later

Synchronous code - What is the problem?

Synchronous I/O or user-interface code

```
let wc = new WebClient()
let data = wc.DownloadData(url)
outputStream.Write(data, 0, data.Length)
```

Blocks thread while waiting

- Does not scale – cannot create 10000 threads
- Blocking user interface – when run on GUI thread
- Simple to write – loops, exception handling etc.

(A)synchronous code

```
var wc = new WebClient();
var html = await wc.DownloadDataTaskAsync(url);
await outputStream.WriteAsync()
```

Easy to **change**, easy to **write**

Can use **loops** and **exception handling**

Scalable – no blocking of threads

Synchronous vs Asynchronous

Task-based Async Pattern (TAP)

Signature

"Async" suffix

Parameters matching synchronous method (no ref or out)

Cancellation/progress parameters only if supported

```
Task<TResult> XxAsync(
    T1 Arg1, T2 Arg2,
    CancellationToken cancellationToken,
    IProgress<TProgress> progress);
```

Returns Task or Task<TResult>

Behavior

Returns quickly to caller
Returned Task must be "hot"
May complete synchronously
Exceptions stored in returned Task (only usage exceptions allowed to escape synchronously)

async & await are about joins, not forks

```
public static Task
PausePrintAsync()
{
    // Requires a thread for all
    // of its processing
    return Task.Run(() => {
        Thread.Sleep(10000);
        Console.WriteLine("Hello");
    });
}
```

```
public static Task
PausePrintAsync()
{
    // Don't consume threads
    // when no threads are needed
    var t = Task.Delay(10000);
    return t.ContinueWith(_ => {

Console.WriteLine("Hello"));
    });
}
```

Task

Synchronous join

```
task.Wait();
```

Asynchronous join

```
await task;
```

Understanding Async

Copy stream example

```
async Task CopyStream(Stream input, Stream output, IProgress<int> progress)
{
    byte[][] buffers = new byte[2][] { new byte[BUFFER_SIZE], new byte[BUFFER_SIZE] };
    int filledBufferNum = 0;
    int readBytes = await input.ReadAsync(buffers[filledBufferNum], 0,
                                                        buffers[filledBufferNum].Length);

    while (readBytes > 0)
    {
        progress.Report(readBytes);

        Task writeTask = output.WriteAsync(buffers[filledBufferNum], 0, readBytes);
        filledBufferNum ^= 1;
        Task<int> readTask = input.ReadAsync(buffers[filledBufferNum], 0,
                                                        buffers[filledBufferNum].Length);
        await Task.WhenAll(readTask, writeTask);
        readBytes = readTask.Result;
    }
}
```

Allocations in async methods

Each async method involves allocations…

- ~~For "state machine" class holding the method's local variables~~
- ~~For a delegate~~
- ~~For the returned Task object~~

Avoided if the method skips its awaits

Avoided if the method skips its awaits,
AND the returned value was "common" …

0, 1, true, false, null, ""

```
public static async Task<int> GetNextIntAsync()
{
    if (m_Count == m_Buf.Length)
    {
        m_Buf = await FetchNextBufferAsync();
        m_Count = 0;
    }
    m_Count += 1;
    return m_Buf[m_Count - 1];
}
```

ConfigureAwait(true) everywhere

```
struct SynchronizationContextRemover : INotifyCompletion
{
        public bool IsCompleted
                => SynchronizationContext.Current == null;
        public void OnCompleted(Action continuation)
        {
                var prevContext = SynchronizationContext.Current;
                try
                {
                        SynchronizationContext.SetSynchronizationContext(null);
                        continuation();
                }
                finally
                {
                        SynchronizationContext.SetSynchronizationContext(prevContext);
                }
        }
        public SynchronizationContextRemover GetAwaiter() => this;
        public void GetResult() { }
}
```

Scalability
is the ability to cope and perform under an increasing workload

Its all about Scalability

```
let httpAsync (url : string) = async {
    let req = WebRequest.Create(url)
    let! resp = req.AsyncGetResponse()
    use stream = resp.GetResponseStream()
    use reader = new StreamReader(stream)
    return! reader.ReadToEndAsync() }

let sites =
    [   "http://www.live.com";"    “http://www.fsharp.org";
        "http://news.live.com";    "http://www.digg.com";
        "http://www.yahoo.com";    "http://www.amazon.com"
        "http://www.google.com";   "http://www.netflix.com";
        "http://www.facebook.com"; "http://www.docs.google.com";
        "http://www.youtube.com";  "http://www.gmail.com";
        "http://www.reddit.com";   "http://www.twitter.com"; ]

sites
|> Seq.map httpAsync
|> Async.Parallel
|> Async.Start
```

Asynchronous Caching

```
static Func<T, Task<R>> MemoizeLazyTaskSafe<T, R>(Func<T, Task<R>> func)
{
    ConcurrentDictionary<T, Lazy<Task<R>>> cache = new ConcurrentDictionary<T, Lazy<Task<R>>>();
    return arg => cache.GetOrAdd(arg, a => new Lazy<Task<R>>(() => func(a))).Value;
}

var client = new HttpClient();
var getData = MemoizeLazyTaskdSafe<string, string>(url =>
{
  Console.WriteLine($"Get String Async {url} - {DateTime.Now.ToString("hh:mm:ss.fff")}");
  return client.GetStringAsync(url);
});

var result_1 = await getData("http://www.google.com");
var result_2 = await getData("http://www.microsoft.com");
var result_3 = await getData("http://www.google.com");
var result_4 = await getData("http://www.microsoft.com");
```

Its all about Asynchronicity !

A different asynchronous pattern
Queuing work for later execution

Concurrent Server Side programming

Reliability and scalability : effort is needed to ensure that applications don't stop working. The throughput needs to scale as new clients start using the service.

Concurrent requests : server-side applications must be designed to enable handling multiple requests concurrently. A program. With a single sequential control flow is bad design.

I/O bound operations : server-side applications exploit large number of I/O operations, which may occupy the largest portion of the request-processing time. This applies to receiving input from the client and sending a response back.

Shared state : servers often exist to enable state sharing among a large number of client. This means that concurrently running processes in a server-side application need to be able to safely access shared states.

Asynchronous Programming on the Server

Async void is only for event handlers!

Principles

- Async void is a “fire-and-forget” mechanism...
- The caller is unable to know when an async void has finished
- The caller is unable to catch exceptions thrown from an async void (instead they get posted to the UI message-loop)

Guidance

- Use async void methods only for top-level event handlers (and their like)
- Use async Task-returning methods everywhere else
- If you need fire-and-forget elsewhere, indicate it explicitly e.g. “FredAsync().FireAndForget()” When you see an async lambda, verify it
- Async all the way!

SynchronizationContext: ConfigureAwait

Task.ConfigureAwait(bool continueOnCapturedContext)

await t.ConfigureAwait(true) // ***default***

Post continuation back to the current context/scheduler

await t.ConfigureAwait(false)

If possible, continue executing where awaited task completes

Implications

- Performance (avoids unnecessary thread marshaling)
- Deadlock (code shouldn’t block UI thread, but avoids deadlocks if it does)

Use ConfigureAwait(false)

Principles

- SynchronizationContext is captured before an await, and used to resume from await.
- In a library, this is an unnecessary perf hit.
- It can also lead to deadlocks if the user (incorrectly) calls Wait() on your returned Task.

Guidance

- In library methods, use "await t.ConfigureAwait(false);"

Configure Context – performance implications

```
async Task MyMethodAsync()
{
    // Code here runs in the original context.
    await Task.Delay(1000);

    // Code here runs in the original context.
    await Task.Delay(1000).ConfigureAwait(continueOnCapturedContext: false);
    // Code here runs without the original
    // context (in this case, on the thread pool).
}
```

Performance implications async/await

Brain Teasers

?????

Symptom: Not Running Asynchronously

```
async void button1_Click(…)
{
    …
    await Task.Run(() => work());
    …
}
```

Problem:
Thinking ‘async’ forks.

Solution:
Use Task.Run around a synchronous method if you need to offload… but don’t expose public APIs like this.

Potential pitfalls when passing around async lambdas
Symptom: Completing Too Quickly

▲ 1 of 12 ▼ ParallelLoopResult Parallel.For(**int fromInclusive**, int toExclusive, Action<int> body)

Executes a for (For in Visual Basic) loop in which iterations may run in parallel.

fromInclusive: *The start index, inclusive.*

```
Parallel.For(0, 10, async i => {
    await Task.Delay(1000);
});
```

Problem:
Async void lambda.

Solution:
Use extreme caution (& knowledge) when passing around async lambdas as void-returning delegates.

▲ 2 of 16 ▼ (awaitable) Task<Task> TaskFactory.StartNew<Task>(**Func<Task> function**)

Creates and starts a System.Threading.Tasks.Task<TResult>.

Usage:
Task x = await StartNew(...);

function: *A function delegate that returns the future result to be available through the S*

```
await Task.Run(
    async () => { await Task.Delay(1000);
});
```

Problem:
Task<Task> instead of Task.

Solution:
Make sure you're unwrapping your Tasks, explicitly (Unwrap) or implicitly.

Symptom: Method Not Completing

```
async void button1_Click(…) {
    await FooAsync();
}

async Task FooAsync()  {
    await
Task.Delay(1000).ConfigureAwait(false);
}
```

Problem:
Deadlocking UI thread.

Solution:
Don't synchronously wait on the UI thread.

Problem:
Deadlocking UI thread.

Solution:
Use ConfigureAwait(false) whenever possible.

New Project
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
.NET Framework 4.5
Sort by:
Default
Search Installed Te
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO

Fractal-Zoom

Fractal-Zoom

```
let rec waiting() = async {
    let! md = Async.AwaitObservable(self.MouseDown)
    // ...
   do! drawing(rc, md.GetPosition(canvas)) }
and drawing(rc:Canvas, pos) = async {
   let! evt = Async.AwaitObservable(canvas.MouseUp,
                                    canvas.MouseMove)
   match evt with
   | Choice1Of2(up) ->
         // ...
         do! waiting()
   | Choice2Of2(move) ->
         // ...
         do! drawing(rc, pos) }
do  waiting() |> Async.StartImmediate
```

Fractal-Zoom

Tasks

1. Convert current functionality to run in parallel
2. Implement Zoom Out

Async Workflow

async workflow

F#

```
let printThenSleepThenPrint = async {
        printfn "before sleep"
        do! Async.Sleep 5000
        printfn "wake up"
    }

Async.StartImmediate printThenSleepThenPrint
printfn "continuing"
```

async workflow

let! – like **`await`** on Task<T> in C#

do! - like **`await`** on Task in C#

return! – like **`return  await`** on Task<T> C#

Anatomy of Async Workflows

```
let readData path : Async<byte[]> = async {
  let stream = File.OpenRead(path)
  let! data = stream.AsyncRead(stream.Length)
    return data }
```

- Async defines a block of code which execute on demand
- Easy to compose

Async Worklfow and parallel programming

Fork-join parallelism : the library function **Async.Parallel** takes a list of asynchronous computations and creates a single asynchronous computation that starts the individual computations in parallel and waits for their completion. (The same as **Task.WhenAll**)

Promise-based parallelism : The F# library primitive for parallel execution is :

Async.StartChild : Async<'T> → Async<Async<'T>>

It takes an async representing a child task and returns an async that represents the completion of the task

Declarative parallelism

Run in parallel and wait for completion

```
var docs = await Task.WhenAll
  (from url in pages select DownloadPage(url));
```

Functional approach

- Works nicely with F# sequences and LINQ

```
let! docs =
 Async.Parallel [ for url in urls -> downloadPage url ]
```

Task-based parallelism

Start task and wait for completion later

```
let! dl1 = Async.StartChild(downloadPage(url1))
let! dl2 = Async.StartChild(downloadPage(url2))
let! html1 = dl1
let! html2 = dl2
```

- C# **Tasks** are started when created
- F# Async started using **StartChild** member

```
var dl1 = DownloadPage(url1);
var dl2 = DownloadPage(url2);
var html1 = await dl1;
var html2 = await dl2;
```

Async and parallel programming

F#

```
let parallel2 (job1, job2) =
    async {
        let! task1 = Async.StartChild job1
        let! task2 = Async.StartChild job2
        let! res1 = task1
        let! res2 = task2
        return (res1, res2) }
```

Cancellation

F#

```
let capability = new CancellationTokenSource()
let tasks =
    Async.Parallel [ getWebPage "http://www.google.com"
                     getWebPage "http://www.bing.com" ]
    |> Async.Ignore

Async.Start (tasks, cancellationToken=capability.Token)

capability.Cancel()
```

C# vs F# Similarity

	C#	F#
Data-parallel	Task.WhenAll	Async.Parallel
Create Task	Task.Factory.Start	Async.StartChild
Continuations	await	let!, do!

C# vs F# Difference

	C#	F#
Running or delayed	Always running	Can be delayed
Cancellation	Explicit pass Cancellation Token	Automatic pass Cancellation Token and check if the operation canceled
Wait inside expression	Can use await inside any expression like await Foo(await Bar())	let! cannot be nested

Async.Catch

```
let asyncTask = async { raise <|
                        new System.Exception(”My Error!") }

asyncTask
|> Async.Catch
|> Async.RunSynchronously
|> function
   | Choice1Of2 result     ->
         printfn "Async operation completed: %A" result
   | Choice2Of2 (ex : exn) ->
         printfn "Exception thrown: %s" ex.Message
```

Lab :
Async Workflow

Concurrent Web crawling

1
2
OS
Scheduler

Lab :
run multiple asynchronous operations in parallel

process the result as complete

TAP combinators

Reliable Tasks with Retry

```
async Task<T> Retry<T>(Func<Task<T>> task, int retries,
                       TimeSpan delay, CancellationToken? cts = null) =>
    await task().ContinueWith(async innerTask =>
    {
        cts?.ThrowIfCancellationRequested();
        if (innerTask.Status != TaskStatus.Faulted)
            return innerTask.Result;
        if (retries == 0)
            throw innerTask.Exception;
        await Task.Delay(delay, cts.Value);
        return await Retry(task, retries - 1, delay, cts.Value);
    }).Unwrap();
```

Retry in action

```
Image image = await Async.Retry(async () =>
                        await DownloadImageAsync("Bugghina001.jpg")
                    , 5, TimeSpan.FromSeconds(2));
```

Reliable Tasks with Otherwise

```
async Task<T> Otherwise<T>(this Task<T> task, Func<Task<T>> orTask) =>
    await task.ContinueWith(async innerTask =>
    {
        if (innerTask.Status == TaskStatus.Faulted) return await orTask();
        return await Task.FromResult<T>(innerTask.Result);
    }).Unwrap();
```

Otherwise in action

```
Image image = await Async.Retry(async () =>
                    await DownloadImageAsync("Bugghina001.jpg")
                .Otherwise(async () =>
                    await DownloadImageAsync("Bugghina002.jpg")),
            5, TimeSpan.FromSeconds(2));
```

Reliable Tasks with Do something and continue

```
Task<T> Tee<T>(this Task<T> task, Action<Task<T>> operation)
{
        operation(task);
        return task;
}

async Task<T> Tee<T>(this Task<T> task, Func<Task<T>, Task> operation)
{
        await operation(task);
        return await task;
}
```

Lab :
Reliable Stock Analysis

Asynchronous Parallel Stock-Tickers

Task Async (TAP) is a Monad

Task async/await is a monad

```
static async Task<R> SelectMany<T, R>(this Task<T> task,
        Func<T, Task<R>> then) => await Bind(await task);

static async Task<R> SelectMany<T1, T2, R>(this Task<T1> task,
         Func<T1, Task<T2>> bind, Func<T1, T2, R> project)
        {
            T taskResult = await task;
            return project(taskResult, await bind(taskResult));
        }

static async Task<R> Select<T, R>(this Task<T> task, Func<T, R> project)
            => await Map(task, project);

static async Task<R> Return<R>(R value) => Task.FromResult(value);
```

Task async/await is a monad

```
static Task<T> Return<T>(T task)=> Task.FromResult(task);

static async Task<R> Bind<T, R>(this Task<T> task, Func<T, Task<R>> cont)
     => await cont(await task.ConfigureAwait(false)).ConfigureAwait(false);

static async Task<R> Map<T, R>(this Task<T> task, Func<T, R> map)
=> map(await task.ConfigureAwait(false));
```

Error handling in functional way

Error handling in functional way

```
module Option =
    let ofChoice choice =
        match choice with
        | Choice1Of2 value -> Some value
        | Choice2Of2 _ -> None

module AsyncOption =
        let handler (operation:Async<'a>) : AsyncOption<'a> = async {
            let! result = Async.Catch operation
            return (Option.ofChoice result)
        }
```

Error handling in functional way

```
let downloadAsyncImage(blobReference:string) : Async<Image> = async {
        let! container = Helpers.getCloudBlobContainerAsync()
        let blockBlob = container.GetBlockBlobReference(blobReference)
        use memStream = new MemoryStream()
        do! blockBlob.DownloadToStreamAsync(memStream)
        return Bitmap.FromStream(memStream)
    }


downloadAsyncImage "Bugghina001.jpg"
|> AsyncOption.handler
|> Async.map(fun imageOpt ->
    match imageOpt with
    | Some(image) -> image.Save("ImageFolder\Bugghina.jpg")
    | None -> log "There was a problem downloading the image")
|> Async.Start
```

Preserving Exception semantic - Result Type

```
type Result<'TSuccess,'TFailure> =
 | Success of 'TSuccess
 | Failure of 'Tfailure

module Result =
        let ofChoice value =
            match value with
            | Choice1Of2 value -> Success value
            | Choice2Of2 e -> Failure e

module AsyncResult =
       let handler (operation:Async<'a>) : AsyncResult<'a> = async {
             let! result = Async.Catch operation
             return (Result.ofChoice result)     }
```

AsyncResult monadic operators

```
module AsyncResult =
    let retn (value:'a) : AsyncResult<'a> =  value |> Ok |> async.Return

    let map (selector : 'a -> 'b) (asyncResult : AsyncResult<'a>) = async {
        let! result = asyncResult
        match result with
        | Ok x -> return selector x |> handler
        | Error err -> return (Error err)    }

    let bind (selector : 'a -> AsyncResult<'b>) (asyncResult : AsyncResult<'a>) = async {
        let! result = asyncResult
        match result with
        | Ok x -> return! selector x |> handler
        | Error err -> return Error err    }
```

Async OOP

Async Constructors – Factory Pattern (1)

```
public sealed class MyClass {
        private MyData asyncData;
        private MyClass() { ... }
        private async Task<MyClass> InitializeAsync()
         {
                asyncData = await GetDataAsync();
                return this;
        }

        public static Task<MyClass> CreateAsync() {
                var ret = new MyClass();
                 return ret.InitializeAsync();
        }
}
public static async Task UseMyClassAsync() { MyClass instance = await MyClass.CreateAsync();
        ... }
```

Async Constructors – Lazy<Task<R>> (2)

```
private static AsyncLazy<T> resource = new AsyncLazy<T>(async () => {
            var data = await GetResource();
            return new MyTinstance(data);
      });

public static async Task UseResourceAsync()
{
       MyTinstance res = await resource;
}
```

Async Constructors – Initialization Pattern (3)

```
public interface IAsyncInitialization
{
        Task Initialization { get; }
}

public sealed class MyFundamentalType : IAsyncInitialization
{
        public MyFundamentalType()
        {
                Initialization = InitializeAsync();
        }

        public Task Initialization { get; private set; }

        private async Task InitializeAsync() => await Task.Delay(100);
}
```

```
var myInstance = new MyFundamentalType();
// the instance is not initialized here!
await myInstance.Initialization;
// now the instance is initialized now.
```

Key Takeaways

- Async void is only for top-level event handlers.
- Use TaskCompletionSource to wrap Tasks around events.
- Use the threadpool for CPU-bound code, but not IO-bound,
- Libraries shouldn't lie, should use ConfigureAwait, should be chunky.

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

-- Edsger Dijkstra