Don't Block on Async Code

4 years ago (Jul 12, 2012) • 143 Comments

This is a problem that is brought up repeatedly on the forums and Stack Overflow. I think it's the most-asked question by async newcomers once they've learned the basics.

UI Example

Consider the example below. A button click will initiate a REST call and display the results in a text box (this sample is for Windows Forms, but the same principles apply to *any* UI application).

```
// My "library" method.
public static async Task<JObject> GetJsonAsync(Uri uri)
{
  using (var client = new HttpClient())
  {
    var jsonString = await client.GetStringAsync(uri);
    return JObject.Parse(jsonString);
  }
}

// My "top-level" method.
public void Button1_Click(...)
{
  var jsonTask = GetJsonAsync(...);
  textBox1.Text = jsonTask.Result;
}
```

The "GetJson" helper method takes care of making the actual REST call and parsing it as JSON. The button click handler waits for the helper method to complete and then displays its results.

This code will deadlock.

ASP.NET Example

This example is very similar; we have a library method that performs a REST call, only this time it's used in an ASP.NET context (Web API in this case, but the same principles apply to *any* ASP.NET application):

```
// My "library" method.
public static async Task<JObject> GetJsonAsync(Uri uri)
{
  using (var client = new HttpClient())
  {
    var jsonString = await client.GetStringAsync(uri);
    return JObject.Parse(jsonString);
  }
}

// My "top-level" method.
public class MyController : ApiController
{
  public string Get()
  {
    var jsonTask = GetJsonAsync(...);
    return jsonTask.Result.ToString();
  }
}
```

This code will also deadlock. For the same reason.

What Causes the Deadlock

Here's the situation: remember from my intro post (/2012/02/async-and-await.html) that after you await a Task, when the method continues it will continue *in a context*.

In the first case, this context is a UI context (which applies to *any* UI except Console applications). In the second case, this context is an ASP.NET request context.

One other important point: an ASP.NET request context is not tied to a specific thread (like the UI context is), but it *does* only allow one thread in *at a time*. This interesting aspect is not officially documented anywhere AFAIK, but it is mentioned in my MSDN article about SynchronizationContext (http://msdn.microsoft.com/en-us/magazine/gg598924.aspx).

So this is what happens, starting with the top-level method (Button1_Click for UI / MyController.Get for ASP.NET):

1. The top-level method calls GetJsonAsync (within the UI/ASP.NET context).
2. GetJsonAsync starts the REST request by calling HttpClient.GetStringAsync (still within the context).
3. GetStringAsync returns an uncompleted Task, indicating the REST request is not complete.
4. GetJsonAsync awaits the Task returned by GetStringAsync. The context is captured and will be used to continue running the GetJsonAsync method later. GetJsonAsync returns an uncompleted Task, indicating that the GetJsonAsync method is not complete.
5. The top-level method synchronously blocks on the Task returned by GetJsonAsync. This blocks the context thread.
6. … Eventually, the REST request will complete. This completes the Task that was returned by GetStringAsync.
7. The continuation for GetJsonAsync is now ready to run, and it waits for the context to be available so it can execute in the context.
8. Deadlock. The top-level method is blocking the context thread, waiting for GetJsonAsync to complete, and GetJsonAsync is waiting for the context to be free so it can complete.

For the UI example, the "context" is the UI context; for the ASP.NET example, the "context" is the ASP.NET request context. This type of deadlock can be caused for either "context".

Preventing the Deadlock

There are two best practices (both covered in my intro post (/2012/02/async-and-await.html)) that avoid this situation:

1. In your "library" async methods, use ConfigureAwait(false) wherever possible.
2. Don't block on Tasks; use async all the way down.

Consider the first best practice. The new "library" method looks like this:

```
public static async Task<JObject> GetJsonAsync(Uri uri)
{
  using (var client = new HttpClient())
  {
    var jsonString = await client.GetStringAsync(uri).ConfigureAwait(false);
    return JObject.Parse(jsonString);
  }
}
```

This changes the continuation behavior of GetJsonAsync so that it does *not* resume on the context. Instead, GetJsonAsync will resume on a thread pool thread. This enables GetJsonAsync to complete the Task it returned without having to re-enter the context.

Consider the second best practice. The new "top-level" methods look like this:

```
public async void Button1_Click(...)
{
  var json = await GetJsonAsync(...);
  textBox1.Text = json;
}

public class MyController : ApiController
{
  public async Task<string> Get()
  {
    var json = await GetJsonAsync(...);
    return json.ToString();
  }
}
```

This changes the blocking behavior of the top-level methods so that the context is never actually blocked; all "waits" are "asynchronous waits".

Note: It is best to apply both best practices. Either one will prevent the deadlock, but *both* must be applied to achieve maximum performance and responsiveness.

Resources

- My introduction to async/await (/2012/02/async-and-await.html) is a good starting point.
- Stephen Toub's blog post Await, and UI, and deadlocks! Oh, my! (http://blogs.msdn.com/b/pfxteam/archive/2011/01/13/10115163.aspx) covers this exact type of deadlock (in January of 2011, no less!).
- If you prefer videos, Stephen Toub demoed this deadlock live (http://channel9.msdn.com/Events/BUILD/BUILD2011/TOOL-829T) (39:40 - 42:50, but the whole presentation is great!). Lucian Wischik also demoed this deadlock (http://blogs.msdn.com/b/lucian/archive/2012/03/29/talk-async-part-1-the-message-loop-and-the-task-type.aspx) using VB (17:10 - 19:15).
- The Async/Await FAQ (http://blogs.msdn.com/b/pfxteam/archive/2012/04/12/10293335.aspx) goes into detail on exactly when contexts are captured and used for continuations.

This kind of deadlock is always the result of mixing synchronous with asynchronous code. Usually this is because people are just trying out async with one small piece of code and use synchronous code everywhere else. Unfortunately, partially-asynchronous code is much more complex and tricky than just making everything asynchronous.

If you *do* need to maintain a partially-asynchronous code base, then be sure to check out two more of Stephen Toub's blog posts: Asynchronous Wrappers for Synchronous Methods (http://blogs.msdn.com/b/pfxteam/archive/2012/03/24/10287244.aspx) and Synchronous Wrappers for Asynchronous Methods (http://blogs.msdn.com/b/pfxteam/archive/2012/04/13/10293638.aspx), as well as my AsyncEx library (http://nitoasyncex.codeplex.com/).

Answered Questions

There are scores of answered questions out there that are all caused by the same deadlock problem. It has shown up on WinRT, WPF, Windows Forms, Windows Phone, MonoDroid, Monogame, and ASP.NET.

Update (2014-12-01): For more details, see my **MSDN article on asynchronous best practices (http://msdn.microsoft.com/en-us/magazine/jj991977.aspx)** or Section 1.2 in my **Concurrency Cookbook (http://stephencleary.com/book/)**.

← Previous Post (/2012/06/arraysegments-library-available.html)   Next Post → (/2012/07/thread-is-dead.html)

143 Comments **Stephen Cleary's Blog** 1 **Login**

Recommend 20 **Share** **Sort by Best**

Join the discussion…

Peter H • 2 years ago

Fantastic post. This helped me out of a conundrum which could have had me stuck for many hours to come.

7 ^ | v • Reply • Share ›

Travis Miller • a year ago

Thanks so much!! This prevented me pulling out what little hair I had left.

4 ^ | v • Reply • Share ›

1antares1 • 2 years ago

Mr. Stephen Cleary, your article has been rapid and great.

My doubts have been clarified. Thanks for taking your time to share the network.

Kindest regards.

4 ^ | v • Reply • Share ›

Thomas Eyde • a year ago

I have an issue with ConfigureAwait(). It may help technically, but it's not very userfriendly. If I have to call ConfigureAwait() almost everywhere, then the defaults are clearly wrong. At some point we're bound to forget that call, and if that's what causes the deadlock, it would be really hard to find. Not to mention that all those calls pollute the code base.

3 ^ | v • Reply • Share ›

Stephen Cleary Site Owner ➦ Thomas Eyde • a year ago

I would say that deadlocks are more caused by sync-over-async behavior. That pattern is unnatural and should be avoided whenever possible. If your code isn't blocking to begin with, then ConfigureAwait just becomes an optimization hint, as it should be.

If you think about it from a higher-level (architectural) perspective, blocking on async code just doesn't make any sense - the blocking is actually preventing the async code from being asynchronous.

There is almost always a way to avoid sync-over-async - the only non-preventable scenarios AFAIK are ASP.NET MVC filters and ASP.NET MVC child actions, both of which have already been addressed in ASP.NET vNext.

That said, I do agree that I wish the default was ConfigureAwiat(false).

1 ^ | v • Reply • Share ›

Thomas Eyde ➦ Stephen Cleary • a year ago

I need more time to wrap my head around this, but my initial reaction is to disagree: It's easier to write and reason about synchronous code. And from a client's perspective, I couldn't care less how a component is implemented, all I care of is when the result is available.

If all my code is synchronous so far, no component should force that to change. I am in charge of my architecture, not a component from nuget.org.

It's a shame there is no safety by design here. Calling Wait() or Result on the same thread should throw an exception. No async code should run on the main thread. I know thread management brings overhead, but we can afford one extra. Anything else is an optimization and should be controlled explicitly.

I've wasted hours to pinpoint that call to Result. It would have been so much better if I got an exception telling me to run that async call on a different thread, or if the runtime just did it.

5 ^ | v • Reply • Share ›

Avraham Y. Kahana • a year ago

Great stuff indeed. Wherever I land upon seeing "Stephen Cleary" I now know it will very likely solve my problem.

3 ^ | v • Reply • Share ›

Guru Rao • 2 years ago

Very helpful.. Thanks for the post!

3 ^ | v • Reply • Share ›

Stephen Cleary Site Owner ➦ Guru Rao • 2 years ago

You're welcome!

1 ^ | v • Reply • Share ›

Ryan B • a year ago

This just save me hours of hair pulling with calling an async repo from a legacy app. Thanks, boss!

1 ^ | v • Reply • Share ›

Rahul P Nath • 2 years ago

Late to come across but still :)
Excellent post. Really elaborate, clear and to the point.

1 ^ | v • Reply • Share ›

Stephen Cleary Site Owner ➦ Rahul P Nath • 2 years ago

Thanks for the kind words, Rahul!

^ | v • Reply • Share ›

alphadork • 2 years ago

Clarity is a beautiful thing! Thanks for the post -- also -- loved the intro to the book. I'll never drink coffee again BTW.

1 ^ | v • Reply • Share ›

Anthony • 2 years ago

Thanks, everything makes much more sense now !

1 ^ | v • Reply • Share ›

Tommy Sadiq Hinrichsen • 2 years ago

Thx, this helped with a problem I had for 2 days now.

1 ^ | v • Reply • Share ›

Ibrahim • 2 years ago

Really helpful post. Thanks.

1 ^ | v • Reply • Share ›

Ed Chavez • 2 years ago

No kidding...it's been 2 years since your post and still, I'm guilty as charged. Dropped on this 2 yr old post in my almost Quixotic quest to see if I can do async in WebPages. Got it to work based on this post - though the "use both" is a mystery still...

I'm getting away with it via ConfigureAwait on the top level request and if there are other async calls that's where "async all the way" occurs. Is this what you meant by "use both"?. Seems to work...

Thanks for posting this - am realizing how much I don't know about (yes, late bloomer). Hopefully your book will change this (just got it). Thanks!

1 ^ | v • Reply • Share ›

Stephen Cleary Site Owner ➦ Ed Chavez • 2 years ago

I recommend using ConfigureAwait for any await that you can. This is usually all await calls *except* the top-level request.

^ | v • Reply • Share ›

yodaflame ➦ Stephen Cleary • a year ago

Not sure this is a good idea. When I do this and an exception is thrown, it takes several seconds for response to come back. When I also use ConfigureAwait(false) on the top level request as well, the exception comes back immediately.

Reply • Share ›

Stephen Cleary Site Owner ➦ yodaflame • a year ago

@yodaflame: You must be seeing something else. ConfigureAwait(false) cannot have an effect like that on execution time.

Reply • Share ›

yodaflame ➦ Stephen Cleary • a year ago

yes you are correct... there was a configure await missing in my chain of calls. I noticed you also replied to my SO post. It would still be nice to understand why you don't recommend using ConfigureAwait(false) at the top level. Also what context is lost by doing this... just the httpcontext or the status of the current class... such as private variables and properties that were set?

Reply • Share ›

Alan • 2 years ago

Thank you for posting this, Stephen, and for taking the time to clearly and concisely explain all of this.

1 • Reply • Share ›

Stephen Cleary • 3 years ago

I assume you mean WCF instead of MVC. It sounds like the root host is still ASP.NET (i.e., your WCF is hosted within ASP.NET).

You can look at your current context by doing a "Debug.WriteLine(SynchronizationContext.Current.GetType().Name)".

To solve the problem, it's best to actually make everything async (WCF in 4.5 does have built-in support for asynchronous implementations, and if you make your server async it won't affect your clients at all). If you can't do that, then you can put in a workaround such as ConfigureAwait. Stephen Toub describes several options here: http://blogs.msdn.com/b/pfxtea...

The disadvantage to ConfigureAwait is that you lose your request context after the first await. So you can't access anything on the HttpContext, or depend on things like culture being set appropriately.

1 • Reply • Share ›

Arya Chuodhury • 5 days ago

Thank you Stephen for the excelency

Reply • Share ›

Ryan Griffith • 5 days ago

When would you want to continue in the same context (i.e. ConfigureAwait(true))?

Reply • Share ›

Stephen Cleary Site Owner ➦ Ryan Griffith • 5 days ago

Any time that the remainder of the `async` method needs the context. E.g., if it accesses UI elements (for UI contexts), or HtppContext.Current (for pre-Core ASP.NET contexts), or implicitly assumes that it resumes on the same thread.

Reply • Share ›

Glushko Dmitrii • a month ago

Hello, according to your post I need to use async all the way down. It's not a problem if it faces page event handlers, because I can use RegisterAsyncTask and create method with returning type Task. But what can i do if I face

overridden method (from external library), and that library doesn't have async method implementation? If I use Result then I get deadlock. Is there any proper workaround to stop 'async all the way down' and get a result without deadlock?
And can someone please explain what's the problem of using Task.Run(() => foo()).Result ? Because foo is calling without context?

• Reply • Share ›

Stephen Cleary Site Owner → Glushko Dmitrii • a month ago

If there's no way around it, then you'll have to block.

The problem with using Task.Run on ASP.NET is one of scalability and efficiency (and, as you pointed out, that the code is run outside the request context). Most (>95%) of people using Task.Run on ASP.NET are not in your situation; they just want to "make it async so it can scale", and don't realize that by using Task.Run, their app is *less* scalable.

In your case, I'd say to first request asynchronous support from your library. And second, use the thread pool if you *have* to: Task.Run(() => foo()).GetAwaiter().GetResult().

More info on "brownfield async": https://msdn.microsoft.com/en-...

• Reply • Share ›

Glushko Dmitrii → Stephen Cleary • a month ago

Thank you very much, almost spent a day of surfing internet and came to the same conclusion. Just wanted to know if my conclusion is correct. As I need my program just to work (no matter async or sync), I will use block for functionality that can't be done async, untill the library will provide async support.

• Reply • Share ›

Jason Shuler • 3 months ago

Thanks for this very helpful post - I do have a question however. I have encountered a situation where I have no control over the library implementation, and I do not have the option of using async/await. For whatever reason the library authors decided to remove all the synchronous versions of the methods, and I am implementing an abstract method on a base class that cannot be defined as async.

It appears that using the following avoids the deadlock:

var t = Task.Run(() => problemObject.ProblemMethodAsync());
t.Wait();

But I have read that this may not be 100% foolproof - some people suggested that it still could be possible for the task to end up on the UI thread and deadlock.

Any ideas?

Thanks!

• Reply • Share ›

Stephen Cleary Site Owner → Jason Shuler • 3 months ago

That approach shouldn't cause a deadlock (assuming that `ProblemMethodAsync` doesn't send updates to the UI thread or anything like that). It *does* assume that `ProblemMethodAsync` can be called on a thread pool thread, which is not always the case.

More info in my async brownfield article: https://msdn.microsoft.com/en-...

P.S. Use GetAwaiter().GetResult() instead of Wait() to avoid the AggregateException wrapper.

• Reply • Share ›

paolo ponzano • 3 months ago

Excuse me, I've read your post and find it really intresting, but a Thing is not clear to me...I've got a WPF application

that uses Catel as MVVM Framework.
I've some services as

public sealed class CurrenciesService : ContainerServiceBase<currency>, ICurrenciesService
{
public override Task<currency> GetItem(int id)
{
return GetItem(x => x.Id == id);
}

protected override Task Init()
{
return PerformInitInternal(InitCacheMessages.STR_GET_COUNTERPARTS,
InitCacheMessages.STR_GET_COUNTERPARTS_FAILED,()=> CommonRepository.GetCurrenciesAsync());
}

protected IEnumerable<t> Items { get; private set; }

---

see more

Reply • Share ›

Stephen Cleary Site Owner → paolo ponzano • 3 months ago
No, that Result call can lead to deadlocks.

I don't think it makes sense to do a service call on a property read. That property should probably be a method. For more info: http://blog.stephencleary.com/...

Reply • Share ›

paolo ponzano → Stephen Cleary • 3 months ago
But I bind that property to a xaml item...in that case I can changeasily it..in behaviors or converters no...

Reply • Share ›

Stephen Cleary Site Owner → paolo ponzano • 3 months ago
Then I recommend you use NotifyTask:
https://github.com/StephenClea...
https://msdn.microsoft.com/en-...

Reply • Share ›

paolo ponzano → Stephen Cleary • 3 months ago
Hello Stephen,I've read your great post on msdn and it clarifies a lot... About notify task can I use it in wpf behavors (I register the notify on attached event?) and converters? (how do I attach it since I've only got covert and convert methods?) should I care about de-registering them since they can lead to memory leak

Reply • Share ›

Stephen Cleary Site Owner → paolo ponzano • 3 months ago
I recommend that you simply *don't* do asynchronous work in a value converter. There is always a better design.

Reply • Share ›

7yhnmju8 • 3 months ago
Thank you very much - your posts around the web have been a great help the last couple of weeks. I think I'm getting there, but I have a question about handling mixed code. I believe this example illustrates it - ideally the database access would also be async but right now is not. Is this code okay for a console or ASP.NET app? Is it helpful, or should I just use synchronous HTTP access? It's async all the way (no Result or Wait), but potentially doing a lot of work in an async method. Is it bad to have any blocking io in an async method?

```
public async Task<IActionResult> ControllerActionAsync()
{
    string stuff = await ServiceMethodAsync();
    return View(stuff);
}

private async Task<string> ServiceMethodAsync()
{
    Task<IList<RestThing>> restTask = GetRestDataAsync(); //async via HttpClient

    IList<DbThing> dbThings = GetDbData(); //legacy sync
```

see more

^ | v • Reply • Share ›

Stephen Cleary Site Owner ➜ 7yhnmju8 • 3 months ago

> Is this code okay for a console or ASP.NET app?

It looks fine to me.

> Is it helpful, or should I just use synchronous HTTP access?

I think it would be helpful to keep the HTTP asynchronous. Making database access async is more debatable - it's only really helpful if the database can scale. But you can generally assume HTTP servers can scale.

> Is it bad to have any blocking io in an async method?

It's not too bad; I cover this in a blog post here: http://blog.stephencleary.com/...

I would say just add a comment to your method stating that it has some blocking in it as well as async code. That way, if it's ever called from a UI app, the dev knows to use Task.Run. It should be called directly (just like you already are doing) when in a Console / ASP.NET / background thread context.

^ | v • Reply • Share ›

7yhnmju8 ➜ Stephen Cleary • 3 months ago

Yes, I remember that post now, too, and it helped inform my reasoning as well. I doubt there will be an UI app, but I understand why that matters. Thanks very much!

^ | v • Reply • Share ›

Fernando Ott • 4 months ago

Thank you Stephen. But this led me to another question. How call async methods on a constructor, when I can't transform this top-level in a async top-level?

^ | v • Reply • Share ›

Stephen Cleary Site Owner ➜ Fernando Ott • 4 months ago

I have an entire blog post on dealing with "async constructors": http://blog.stephencleary.com/...

^ | v • Reply • Share ›

Chiranjib • 5 months ago

Awesome post . And I am eager to discuss a problem that I faced.
On the web service side here are four layers : controller,business and DAL and DAL talks to DB.

Now from service side I want to invoke the three methods like

public async Task<groupmembershipvalidationoutput> validateDetails(myObject _input1)

{

Data.Upload.UploadDetails gd = new Data.Upload.UploadDetails();

var firstTask = gd.myMethod1(_input1);
var secondTask = gd.myMethod2(_input1);
var thirdTask = gd.myMethod3(_input1);

Here is how DAL methods look like

public async Task<ienumerable<aout>> myMethod1(myObject gmvi)
{
Repository rep = new Repository();

see more

Reply • Share ›

Stephen Cleary Site Owner → Chiranjib • 5 months ago

It's not clear from your description (or SO question) what exactly you're seeing. The VS debugger will take special actions to allow you to "step" naturally through async code, which is why I asked about whether you're stepping through your methods.

Regardless, as long as your DB queries are truly asynchronous, then they should all be running concurrently. If you have any warnings saying "this async method lacks await operators and will run synchronously", then it's not actually asynchronous and won't be concurrent.

Reply • Share ›

Chiranjib → Stephen Cleary • 5 months ago

I put the breakpoints .. for each of the methods .. once in service class. then in the DAL class and then finally in the DB repository class. But as far as I could see once a method is called and the method call break point is hit , it immediately goes to the next DAL layer break point for that corresponding method. But accordingly , should not all the serviceclass methods breakpoints be hit one by one , and then the Debugger move to the DAL layer for any of the methods ?

And , the main concern is , Why is await.WhenAll(task1,task2,taks3) immediately returning to the invoking controller ?

I have not yet got the results .. and hence I am inevitably getting null as the controller returns results back to the UI side ? How can I avoid it ? Wait for all the tasks to asynchronously and then wait for all the results to come back and then only return the consolidated result back to Controller , so the Controller now can return successfully to UI ?

Please help.

Reply • Share ›

Stephen Cleary Site Owner → Chiranjib • 5 months ago

Once an operation is in flight, it can complete at any time. It's entirely possible that by running in the debugger and stopping at breakpoints, you're slowing down the application sufficiently that the operations *appear* serialized.

await will immediately return - of course it will, that's the entire *point* of await. It sounds like you may be missing an await in your calling code. Every task returned by any methods in your code should be awaited at some point - if you find one that's not, then that's probably your problem. (A good first step is to turn on warnings as errors, since the compiler does its best to ensure you always await your tasks. It's not perfect, but it'll catch a lot of them).

Reply • Share ›

Shamal Badhe • 5 months ago

Hello Stephen,
I'm stuck in this issue: This is my coding structure.

public static async Task<string> A()

```
{
.....
await Test1.Get();
....
await Test2.Post();
.......
await Test3.Send().ConfigureAwait(false);
}

public class Test1{
public static Get(){
.......
await Inner.Pass();
.......
```

see more

Reply • Share ›

Stephen Cleary Site Owner → Shamal Badhe • 5 months ago

There's nothing in the code you posted that would cause a deadlock. And the "thread has exited" statements don't mean anything useful in this case; you can just ignore them.

I recommend that you:
1) Reduce the code causing the deadlock down to the minimum possible code that still causes the deadlock.
2) Post a question on Stack Overflow with *all* of the code necessary to reproduce the issue.
3) Send me a link to the question.

Reply • Share ›

Shamal Badhe → Stephen Cleary • 5 months ago

Thank you Stephen!
I reduced the code as well that was causing deadlocks. I fixed it.
Can you please tell me in what situation, await method does not return the value? It executes very well. But after completing the function, doesn't return the value to its parent method.

Reply • Share ›

Stephen Cleary Site Owner → Shamal Badhe • 5 months ago

If await returns at all (i.e., it doesn't deadlock), it will return the result of the task (if there is any).

You can think of `await` as "unwrapping" tasks. When a result is placed on a task, then it is completed and the `await` gets that value.

If you `await` a plain (nongeneric) `Task`, then there is no result type at all.

Reply • Share ›

Load more comments

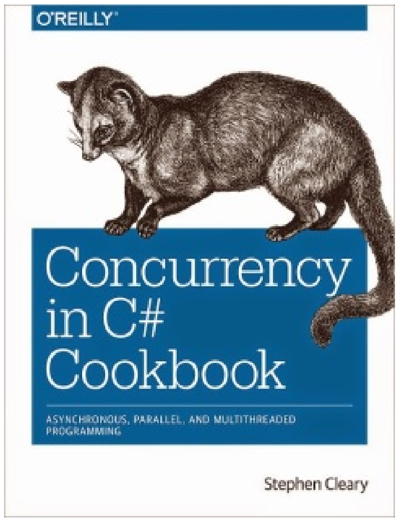
Subscribe Add Disqus to your site Add Disqus Add Privacy

About Stephen Cleary

Stephen Cleary is a Christian (http://stephencleary.com/god/), husband, father, and programmer living in Northern Michigan.

Microsoft® MVP Most Valuable Professional (http://mvp.microsoft.com/en-us/mvp/Stephen%20Cleary-5000058)

My book


O'REILLY
Concurrency in C# Cookbook
ASYNCHRONOUS, PARALLEL, AND MULTITHREADED PROGRAMMING
Stephen Cleary
(http://stephencleary.com/book/)

Available from O'Reilly (http://tinyurl.com/ConcurrencyCookbook) or Amazon (http://tinyurl.com/ConcurrencyCookbookAmazon).

Advertisement

Popular Posts

Async/await Intro (/2012/02/async-and-await.html)