25 SECRETS FOR *FASTER* ASP.NET

JEFFREY RICHTER
JOHN ROBBINS
DAVID CONLIN
SHMUEL ENGLARD
RYAN RILEY
CHRIS ALLEN
ROBERT HAKEN
MITCHEL SELLERS
NIALL MERRIGAN
CHRIS HURLEY
RAGHAVENDRA
MATT LEE
JP TOTO
MICHAEL WILLIAMSON
TIAGO PASCOAL

Foreword

When we launched the predecessor to this book, 50 Ways to Avoid, Find and Fix ASP.NET Performance Issues, we had no idea what a hit it would be. More than 15,000 of you have got a copy, it's been featured on www.asp.net, and it took Twitter by storm.

Rising adoption of MVC 4 and Web API means more new chances for performance improvements than you can shake a stick at, so this book adds (among other things) a fistful of new tips for those technologies, all from members of the ASP.NET community. I'd like to thank them all for their excellent contributions.

Last time, I said that between us we could make ASP.NET applications run faster than Usain Bolt with cheetahs for shoes. We're not quite there yet, but I hope all of you find a few tips more here to put a gazelle-like spring in your applications' step.

Michaela Murray

dotnetteam@red-gate.com

redgate

Contents

Want to build scalable websites and services? Work asynchronously

Jeffrey Richter
www.wintellect.com

One of the secrets to producing scalable websites and services is to perform all your I/O operations asynchronously to avoid blocking threads.

When your thread issues a synchronous I/O request, the Windows kernel blocks the thread. This causes the thread pool to create a new thread, which allocates a lot of memory and wastes precious CPU time. Calling *xxxAsync* method and using C#'s *async/await* keywords allows your thread to return to the thread pool so it can be used for other things. This reduces the resource consumption of your app, allowing it to use more memory and improving response time to your clients.

Where are your custom performance counters?

John Robbins

@JohnWintellect, www.wintellect.com

Performance counters are a great way to watch how you are using .NET, IIS, the database, and much more on the operating system. However, the real benefits of performance counters come when you start using them to record data that's unique to you, such as how many logins you've had, what's the average time for your web service to process a specific request, and anything else you can imagine.

With .NET's excellent PerformanceCounter class doing the work for you, creating performance counters is so simple a manager could do it. By spending no more than an hour on planning and implementation, you can get a ton of actionable information out of your application in production. Performance counters are cheap, easy, and you can never have enough.

RavenDB

David Conlin

Red Gate, agilelikeacat.com

RavenDB has built in protection against the select n+1 problem and also forces you to think about paging early on. It's really good at making you build database performance in from the ground level.

Don’t call AsEnumerable on a collection before using LINQ

Shmuel Englard

@shmuelie, blog.englard.net

When you’re using an ORM that has LINQ to SQL, such as Entity Framework, do not call AsEnumerable on the collection before using LINQ, even if that gives you options that are easier to work with.

If you do, it means your LINQ query is run client-side, rather than converted to SQL and performed on the database server.

A selection of tips from

Ryan Riley
@panesofglass

5

Never call .Wait() or .Result on a Task

Never call .Wait() or .Result on a Task within your application; it can bring your server crashing down. It shouldn't be necessary to mention this, but so many examples do this that it's worth reiterating.

6

Throwing HttpResponseExceptions

This is a very handy way to skip to the end of a processing chain when something goes wrong. Sure, you can return an *HttpResponseMessage*, but it will pass through the filters and handlers on its way out. Throwing an *HttpResponseException* skips all that and goes right to the end. Use it with the same care you would use in throwing any exception.

Web API tracing

Web API integrates tracing, so as to make it very easy to know what's going on throughout the lifetime of your services. Microsoft has released an add-on package to enable tracing of Web API using *System.Diagnostics*. You can also find adapters for NLog and log4net on NuGet. This is really helpful for tracking down unexpected responses or unhandled exceptions, especially in production applications. For the best performance, of course, you should use Event Tracing for Windows (ETW).

You can find a sample project showing how this can work with Web API in the samples here.

Message Handlers

If you only have to handle simple cases and don't need complex routing, you can handle a given route directly by inheriting from *DelegatingHandler* and setting this custom handler as the last parameter of **config.Routes.MapHttpRoute**. This completely bypasses the filter pipeline and requires a lot more manual work, but it also bypasses all the Reflection-based controller and action lookups. Here's an example: https://gist.github.com/panesofglass/5831674.

You can also share common logic between some of the controllers by using this approach and manually hooking up the controller dispatcher, as in https://gist.github.com/panesofglass/5831674. This is slightly better than the filter approach because you can return a response before all the Reflection-based lookups occur, although even the default Reflection-based lookups are fast, because they're cached.

Database access

Most examples of Web API use either Entity Framework directly or show some abstractions such as *Repository* and/or *UnitOfWork*, which introduce a number of classes, as well as strong typing. That's all good. However, in many Web API cases you're simply exposing, and possibly transforming, data from a data store. You are probably reading in values from a query string, JSON, or XML, and manipulating SQL. All of these are strings. While you can certainly use a serializer to work with inputs and use casting to correctly validate data types, in many cases you can forego the cost of boxing/unboxing or serializing/deserializing by using dynamics. I have a gist that shows how to do this using Dapper (with Async support):

https://gist.github.com/panesofglass/5212462. Use it as a way of eking out extra performance, rather than a standard approach.

When you're profiling, prefer accuracy to detail

Chris Allen
Red Gate, www.facebook.com/chrissy.f.allen.1

There is always a trade-off between accuracy and detail, and before you start optimizing you want to be very sure of your data. Sampling techniques tend to be more accurate than instrumentation techniques.

If you feel you really must get line-level detail, do it as a second pass after you've narrowed down your most expensive call stacks.

Make the most of connection pooling by closing SQLConnection as soon as possible

Robert Haken

@RobertHaken, knowledge-base.havit.cz

Close or *Dispose* your *SqlConnection* as quickly as possible to take advantage of connection pooling in ADO.NET. Closed connections return to the connection pool, where they remain cached, so you won't need to create a new connection. Take advantage of the **using** statement to restrict the scope of your SqlConnections for the short time you need them.

OutputCache

Robert Haken
@RobertHaken, knowledge-base.havit.cz

Use output caching whenever your rendered HTML does not vary between requests, or if you have only a few variants. If you need isolated updates, consider using the *Substitution* control, or apply selective output caching by breaking your page into several *UserControls*, each with its own *OutputCache*.

13

Use ConfigureAwait to avoid thread hopping, especially in library code

Mitchel Sellers
@mitchelsellers

One of the most amazing thing about using *async/await* keywords for async and background processes is that you don't have to do anything special to be able to interact with the UI thread.

For example, if you trigger an async operation on the UI thread, then interact with a *TextBox* after the code returns, the .NET framework will automatically marshal the continuation back to the UI thread for you behind the scenes.

This is often very helpful, because you need to interact with the UI thread, but there are other times where you really are thread agnostic. In those cases it is important to use *ConfigureAwait* to set *ContinueOnCapturedContext* to false. This ensures that the .NET runtime doesn't have to go through the effort of resuming your method on the thread that called it. In general, this prevents a lot of back-and-forth thread hopping, but in some cases it can be even more helpful.

Consider this example using ConfigureAwait:

```
byte [] buffer = new byte[0x1000];
int numRead;
while((numRead = await
source.ReadAsync(buffer, 0, buffer.Length).ConfigureAwait(false)) > 0)
{
await source.WriteAsync(buffer, 0, numRead).ConfigureAwait(false);
}
```

This code reads an input source in small blocks. If we were working with a large input stream, we could easily have 10, 20, or even more different ***await*** cycles. Each cycle through is at least two async calls, with two sets of callbacks to the calling thread. By using *ConfigureAwait(false)*, the total time to execution is much lower and we truly get the best performance out of async.

Be careful of variable allocations

Mitchel Sellers

@mitchelsellers

Inside async methods, .NET will create a state machine behind the scenes to lift out local variables for you. That way, when the method resumes, all the values are still there. It's a fantastic feature, but at it's not yet smart enough to tell if you still need a particular variable.

Consider this code example:

```
var today = DateTime.Now;
var tomorrow = today.AddDays(1);
await MyTaskHere();
Console.WriteLine(tomorrow);
```

In this example we have two **`DateTime`** variables declared - yes, it's a trivial example, but it proves the point. After the await, we only need to use the "tomorrow" value. However, the state machine will lift out both the today and tomorrow variables. In this example, the object that is used to retain state for the async operation is larger than necessary. This means more variable declarations, and eventually this can lead to additional GC cycles, and so on.

To avoid this, and reduce the size of the state-tracking object, you could re-write the code like so:

```
var tomorrow = DateTime.Now.AddDays(1);
await MyTaskHere();
Console.WriteLine(tomorrow);
```

By being smart with the variables that you have inside an async method, you can help control the size of the state-tracking object.

How to stress test your public facing web application using the cloud (or without it)

Niall Merrigan
@nmerrigan

1. Simple HTTP calls

Running Web Capacity Analysis Tool (WCAT) on Azure allows you to create multiple instances of a VM from different geographic locations, and thoroughly stress the application to find areas of poor performance. You can then track load

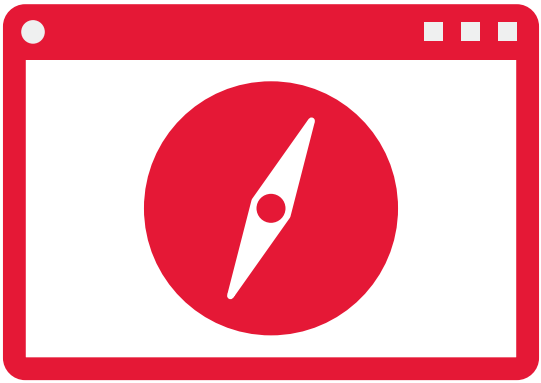
If your site is internal, you can still use WCAT to simulate load. It's advisable to run it from different machines to give the best results.

You can download WCAT from IIS.NET:

2. Coded UI Tests

The more adventurous among you can also create Coded UI Tests (CUITs) and implement them on Azure. CUITs can interact with the application and test parts that a simple HTTP call cannot.

To make this work you need to have Visual Studio Premium, a UIMap to do the work on, and an open desktop on the VM in the cloud. An extra advantage is that you can test your application from different geographic locations and see how it performs when doing actual scripted UI commands.



16

Using the keyword await doesn't make the work asynchronous

Chris Hurley
Red Gate

The *async/await* pattern makes it easy to write code that depends on work that's done asynchronously, but it doesn't turn synchronous code into asynchronous code.

If you await an *async Task* method that does some work and returns a value, the work will be done synchronously – the original method is waiting for a *Task* object to be returned which it can then await on, but the only *Task* it gets is one which is wrapped around the return value and set to "completed". This is useful if you are chaining *async* methods, but in this case it is confusing.

By the time the awaiting method sees the *Task*, it's already complete, so the code continues to execute synchronously. If you want to do work asynchronously, you need to pass back a *Task* object representing the work in progress. One way of doing this is by using *Task.Run()*.

17

Don't use async/await for short methods

Chris Hurley Red Gate

Async/await is great for avoiding blocking while potentially time-consuming work is performed, but there are overheads associated with running an *async* method: the current execution context has to be captured, there may be a thread transition, and a state machine is built through which your code runs. The cost of this is comparatively negligible when the asynchronous work takes a long time, but it's worth keeping in mind.

Avoid using *async/await* for very short computational methods or having await statements in tight loops (run the whole loop asynchronously instead). Microsoft recommends that any method that might take longer than 50ms to return should run asynchronously, so you may wish to use this figure to determine whether it's worth using the *async/await* pattern.

This doesn't apply when using the framework-provided async I/O methods, which are designed to return synchronously where possible and generally resume on the same thread that was handling the original request. Therefore it's fine to use *async/await* whenever potentially-blocking I/O occurs, allowing your ASP.NET application to process concurrent requests more efficiently.

18

Turn off Change Tracking in Entity Framework

Raghavendra
@vamosraghava

If you have to query a database for some read-only data in Entity Framework, make sure that Change Tracking is turned off, so you're not tracking individual objects or object graphs whose state won't change. Use some code along these lines to do it: Context.MyCollection.AsNoTracking().where(x=>x.Id);

For example:

```
using (var context = new BloggingContext())
{
        // Query for all blogs without tracking them
        var blogs1 = context.Blogs.AsNoTracking();
        // Query for some blogs without tracking them
        var blogs2 = context.Blogs
                .Where(b => b.Name.Contains(".NET"))
                .AsNoTracking()
                .ToList();
}
```

AsNoTracking() is an extension method defined on *IQueryable<T>*, so you'll need to import the *System.Entity.Data* namespace. For more details, to understand the performance gains of *AsNoTracking()*, have a look at

http://msdn.microsoft.com/en-us/library/cc853327.aspx and

http://blog.staticvoid.co.nz/2012/4/2/entity_framework_and_asnotracking

Always use compiled queries in Entity Framework

Using compiled queries on top of *ObjectContext* or *DBContext* can significantly improve application performance, because you save much of the time that Entity Framework would have to spend compiling your query to SQL.

20

Diagnosing JavaScript memory leaks with Chrome Dev tools

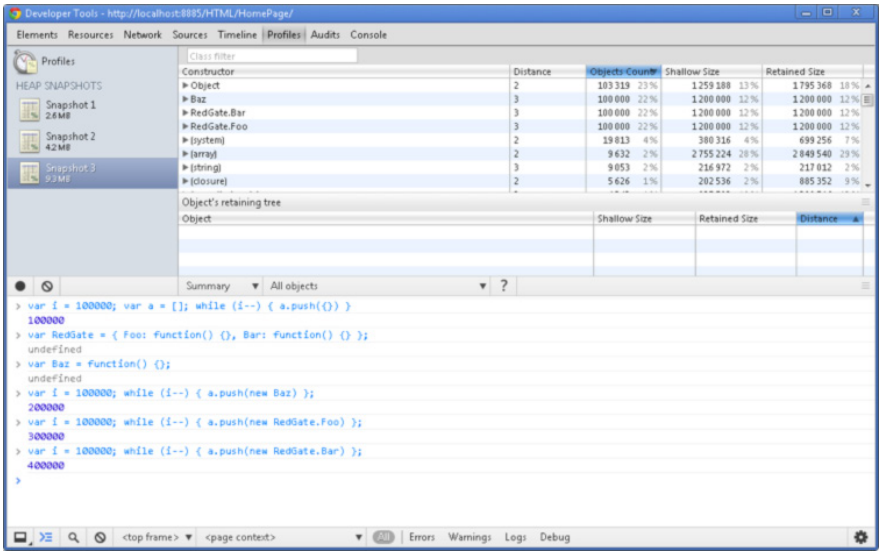
Matt Lee

@thatismatt, github.com/thatismatt

A heap snapshot in Chrome Dev Tools aggregates your objects by type, so anonymous objects are grouped together, which makes it tricky to diagnose a memory leak's cause.

A neat workaround is to construct your objects with a named constructor function. If you namespace these functions too, you end up with a set of objects with names like e.g. RedGate.Foo, where once all you had was a load of "Object"s. Now you can easily filter by type, find out if it really is your code that's causing a memory leak, and hopefully pinpoint which type is the problem.

Here's an illustration of the sort of results you can get, using a toy example where we insert objects into an array but never remove them:



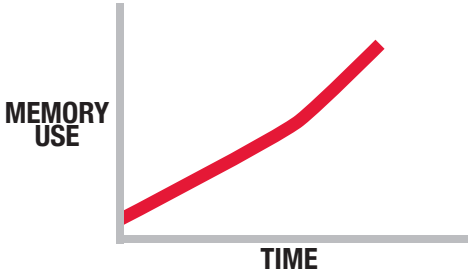
21

Monitoring memory consumption over time

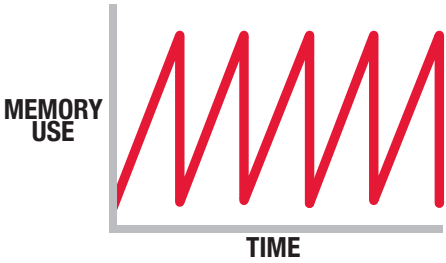
Matt Lee

@thatismatt, github.com/thatismatt

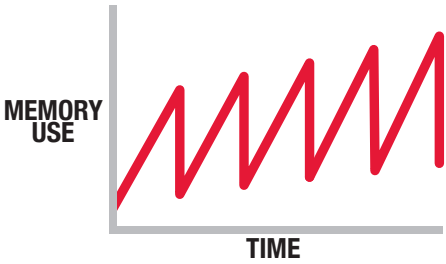
When you're hunting for a memory leak, it's easy to fall into the trap of not profiling for long enough. Consider this curve:



On the face of it, it looks like we might have a memory problem. But of course, garbage collection only happens periodically, so it's perfectly legitimate for browser memory usage to increase over time. What matters is the pattern you see when garbage collections occur. You expect a sawtooth pattern, a bit like this:



If the sawtooth stays flat, as in the example above, you probably don't have a leak. On the other hand, if it's increasing, as in the example below, you probably do:



The important point is to make sure you profile long enough to see the whole curve. Otherwise, you might mistakenly believe you have a memory leak, and misidentify the cause as objects that will be garbage collected.

22

Use JValue in JSON.Net to parse complex JSON objects that you don't have POCO types for

JP Toto
@jptoto

When parsing the JSON response from an API, it's not always easy to model the response perfectly in C#. Fortunately, you may only need part of the response. This is where the static JValue class in JSON.Net comes in handy.

If you've got your response string, you can parse it into a dynamic type and then access the object in whatever way you need, so long as you know its structure:

```
using Newtonsoft.Json.Linq;
dynamic json = JValue.Parse(response_string);
foreach (dynamic something in json)
{
        string name = something.name;
        int count = Convert.ToInt32(something.total);
}
```

It's quite handy and much easier than trying to model a giant POCO after some large JSON response.

Cache JavaScript and CSS permanently

Michael Williamson
mike.zwobble.org

Cache static content such as JavaScript and CSS files permanently, and change the URL when the contents change. For bonus points, automatically generate the URL based on the hash of the content, so you don't need to remember to update the URL manually. For instance, make */static/hash-of-the-content/js/main.js* an alias for */static/js/main.js* using URL Rewrite or somesuch.

Load external JavaScript content asynchronously

Michael Williamson
mike.zwobble.org

Otherwise, if an external site is loading slowly, your own page will stall until the external site finally finishes loading.

If you're including something that can't be loaded asynchronously, for instance, an advert snippet that uses document.write, put it in an iframe instead.

Profile, don't speculate!

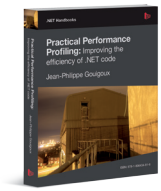
Tiago Pascoal
pascoal.net

Performance is something you should be concerned with from the start, by following best practices and general guidelines, rather than focusing on nitty gritty details. However, there often comes a time where you need to optimize your code to make it faster.

It's tempting to rely on your gut and focus on spots of code that you think are slow, but if you do that you run the risk of speeding up code that doesn't make much difference to overall performance. After all, even if you double the performance of code that only executes for 1% of overall time, the end result will be negligible.

Don't speculate. Use code profilers to measure where time is being spent (either optimize for speed or for memory use), so you can focus your energy on the optimizations that will have the greatest impact on the overall performance of your system.

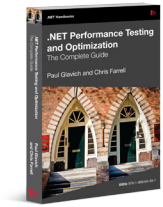
More free eBooks from Red Gate



Practical Performance Profiling: Improving the efficiency of .NET code

by Jean-Philippe Gouigoux

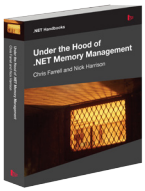
Theory and practical skills to analyze and improve the performance of .NET code. Gouigoux guides the reader through using a profiler and explains how to identify and correct performance bottlenecks.



.NET Performance Testing and Optimization

by Paul Glavich and Chris Farrell

A comprehensive and essential handbook for anybody who wants to set up a .NET testing environment and get the best results out of it, or learn effective techniques for testing and optimizing .NET applications.



Under the Hood of .NET Memory Management

by Chris Farrell and Nick Harrison

Chris Farrell and Nick Harrison take you from the very basics of memory management, all the way to how the OS handles its resources, to help you write the best code you can.

Tools from Red Gate

ANTS Performance Profiler

Identify bottlenecks and optimize the performance of your application.

ANTS Memory Profiler

Find memory leaks and optimize the memory usage of your application.

.NET Reflector

Browse, analyse, decompile, and debug your .NET code.

SmartAssembly

.NET obfuscator to protect your IP; plus, Error Reporting functionality to help you ship stable software by getting early user feedback.

.NET Demon

.NET Demon compiles your code continuously, so you see errors as soon as they are introduced.