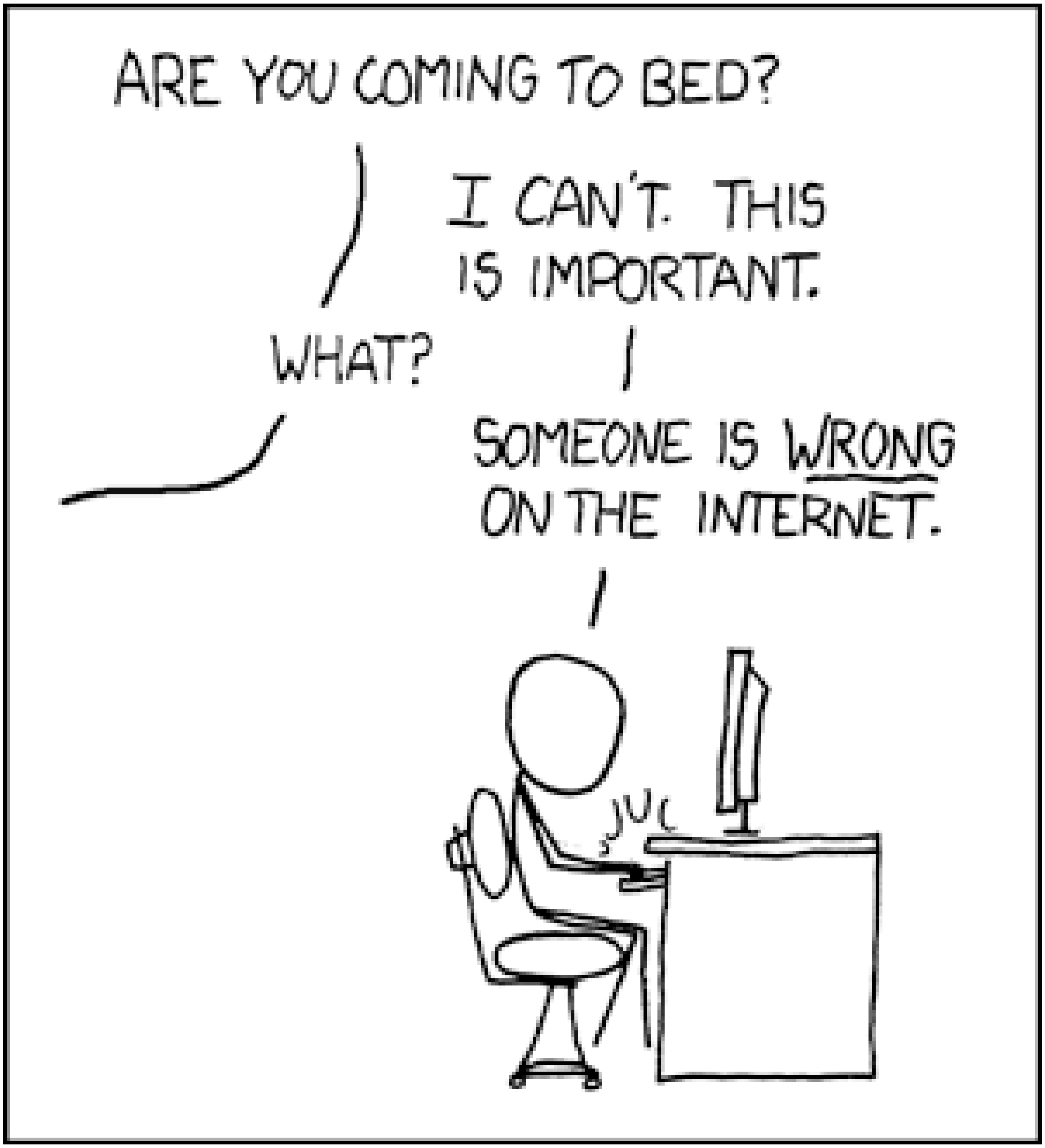
2018-03-22 / BENCHMARKING

Is WCF faster than ASP.NET Core? Of course not! Or is it?

I was casually browsing Reddit when I came across a comment that triggered me. I'm paraphrasing[1], but the gist of it was that 'WCF is faster than Web API or ASP.NET Core'. Surely, that was a mistake.


ARE YOU COMING TO BED?
I CAN'T. THIS IS IMPORTANT.
WHAT?
SOMEONE IS WRONG ON THE INTERNET.

"Duty Calls" by xkcd is licensed under CC BY-NC 2.5.

Boy, I'll show them. What are they claiming, actually? 'The response times of a WCF service are much lower than those of ASP.NET Web API or ASP.NET Core MVC.' Pfft. I'll just write a small benchmark using trusty ol' BenchmarkDotNet. Stand up a local web server, measure how long it takes to create a request, send it, deserialize it, generate a response, send that back, and deserialize the response. One method will use WCF, the other will use

ASP.NET Web API. To quote a colleague of mine: ‘how hard could it be?’

> *If you’re only interested in the complete picture, take a look at the full table of results and the graphs.*

First steps

I’m sending 100 very simple objects (a single property that is a GUID) to the API and it’s sending 100 items back. Writing the benchmark wasn’t hard, but processing the outcome was, kind of. WCF **was** faster than ASP.NET Web API.

METHOD	MEAN
Wcf	830,1 μs
WebApi	2 614,2 μs

And not just a little bit faster; WCF is putting Web API to shame by taking less than a third of the time. Alright, but Web API is a pretty obsolete technology. Surely the new and shiny ASP.NET Core MVC will do much better. Right?

METHOD	MEAN
Wcf	830,1 μs
WebApi	2 614,2 μs
AspNetCore	**2 524,8 μs**

Well, it’s a *little* faster, but WCF still takes less than a third of the time.

Different formats

What's so different between WCF and the other two options? Well, there is a pretty obvious difference: WCF is serializing data to and from XML (SOAP, to be precise), while for Web API and ASP.NET Core MVC, I was defaulting to JSON. What if I force the APIs to use XML?

METHOD	MEAN
Wcf	830,1 μs
WebApiJson	2 614,2 μs
AspNetCoreJson	2 524,8 μs
WebApiXml	**1 982,7 μs**
AspNetCoreXml	**1 933,5 μs**

There's a definite improvement there. WCF is still a lot faster, but at least this shows we can get better results by picking a different serializer.

MessagePack

So what's the fastest serializer we can find? According to their own claims, MessagePack is a small and fast format, and Yoshifumi Kawai has written a high-performance MessagePack serializer for .NET that beats protobuf-net, the default Protobuf serializer for .NET. Protobuf was designed to be easy to serialize and is known for its performance. Let’s see.

METHOD	MEAN
Wcf	830,1 µs
WebApiJson	2 614,2 µs
AspNetCoreJson	2 524,8 µs
WebApiXml	1 982,7 µs
AspNetCoreXml	1 933,5 µs
WebApiMessagePack	**1 615,7 µs**
AspNetCoreMessagePack	**1 483,8 µs**

It’s going in the right direction. But can we go any faster? Probably not without extensive customization. However, these benchmarks have all been about serializing and deserializing 100 very simple objects. In the real world, objects are usually more complex than just a single GUID property.

Larger objects

Let’s try a more complex object.

```
public class LargeItem
{
    public Guid OrderId { get; set; }
    public ulong OrderNumber { get; set; }
    public string EmailAddress { get; set; }
    public Address ShippingAddress { get; set; } = new A
    public Address InvoiceAddress { get; set; } = new Ad
    public DateTimeOffset RequestedDeliveryDate { get; se
    public decimal ShippingCosts { get; set; }
    public DateTimeOffset LastModified { get; set; }
    public Guid CreateNonce { get; set; }
    public List<OrderLine> OrderLines { get; set; }
}

public class OrderLine
{
    public string Sku { get; set; }
    public int Quantity { get; set; }
    public string Product { get; set; }
    public decimal Price { get; set; }
}

public class Address
{
    public string Name { get; set; }
    public string Street { get; set; }
    public string HouseNumber { get; set; }
    public string PostalCode { get; set; }
    public string City { get; set; }
    public string Country { get; set; }
}
```

This is an order with order lines, copied from one of my other projects. There is a reasonable number of properties there, so let's see how it goes.

METHOD	MEAN
LargeWcf	9 890,2 μs
LargeWebApiJson	25 425,6 μs
LargeAspNetCoreJson	40 312,9 μs

METHOD	MEAN
LargeWebApiXml	16 193,5 μs
LargeAspNetCoreXml	36 767,1 μs
LargeWebApiMessagePack	8 834,9 μs
LargeAspNetCoreMessagePack	8 813,8 μs

Bingo. When serializing and deserializing 100 'large' items, MessagePack on either ASP.NET Web API or ASP.NET Core MVC beats WCF by a small margin.

#itdepends

As usual, the answer to the question 'which is faster' is: it depends. When performance is absolutely critical, and your service doesn't have to be a public API that anyone can consume, WCF is probably your best bet, up to a certain point. When you need to communicate large amounts of complex data, MessagePack on top of ASP.NET Core MVC is probably a better solution. Also worth considering is the developer-friendliness, where WCF doesn't score very high marks.

What about JSON? The default library for working with JSON in .NET is Newtonsoft.Json. It's great when you have to have absolute control over how your JSON looks, but it's by far the slowest option I've examined. Can we have our cake and eat it too?

Utf8Json

It turns out that, yes, we can. Sort of. Yoshifumi Kawai, who you'll remember as the author of the very fast MessagePack serializer for .NET, has also written a performance-oriented JSON serializer for .NET called Utf8Json. It's not as customizable and it doesn't support DOM operations, but man, is it fast.

METHOD	MEAN
LargeWcf	9 890,2 μs
LargeWebApiJson	25 425,6 μs
LargeAspNetCoreJson	40 312,9 μs
LargeWebApiXml	16 193,5 μs
LargeAspNetCoreXml	36 767,1 μs
LargeWebApiMessagePack	8 834,9 μs
LargeAspNetCoreMessagePack	8 813,8 μs
LargeWebApiUtf8Json	**13 787,9 μs**
LargeAspNetCoreUtf8Json	**13 961,1 μs**

It's not as fast as WCF, and definitely not as fast as MessagePack, but it easily takes less than half the time that Newtonsoft.Json does. It does this by avoiding memory allocations (similar to the MessagePack serializer) and by treating JSON as a binary format; it doesn't serialize to a string which is then converted into bytes, but instead it serializes straight to UTF-8 bytes[2].

Conclusion

As I said before, it depends. If round-trip time performance is more important than pretty much anything else, but you still want to use a managed language, use WCF. If you care about performance, but also about legible and easy to use code, or you want to create a proper REST-ful service, use MessagePack or Utf8Json on top of ASP.NET Core MVC or ASP.NET Web API.

The code I've used to benchmark these libraries is available on GitHub. Feel free to play around with it, and maybe find even better options.

Data

Below you'll find the 'raw' output from the benchmark I've run. Some annotations:

- A method prefixed with 'Small' means it serializes and deserializes a small class with only a single GUID property. 'Large' means it serializes and deserializes the large class described above.
- All WCF methods are hosted using `WebServiceHost` ;
- All WCF methods use a `ChannelFactory` client, while the Web API and ASP.NET Core MVC methods use `System.Net.HttpClient` [3];
- `WcfText` uses a `BasicHttpBinding` with the message encoding set to `Text` ;

- `WcfWebXml` and `WcfWebJson` use a `WebHttpBinding`, with, respectively, XML or JSON as the message format;
- `JsonNet` methods use Newtonsoft.Json to serialize objects;
- I’ve included ‘0 items’ as an indication of how fast the client and web server/framework are;
- The ‘P95’ column represents the 95th percentile value of the response time;
- The ‘Gen 0’, ‘Gen 1’ and ‘Gen 2’ columns represent the number of garbage collections per 1 000 invocations;
- The ‘Allocated’ column represents the total amount of memory allocated for each invocation;

Some interesting data points that jump out:

- At 0 items, ASP.NET Core makes significantly less allocations, and as a result, it is the only option that results in *no* Generation 1 (or higher) allocations;
- Even with 0 items, the options using Newtonsoft.Json are a lot slower than any of the other options. There must be a lot of initialization happening;
- For larger item sizes, Utf8Json is frequently allocating the most memory of any option, but it still manages to be pretty fast. That just goes to show - avoiding memory allocation isn’t the only thing you need for speed;

- ASP.NET Core's XML serializer is a lot slower than Web API's - this is particularly visible for large items;
- WCF does *very well* in terms of allocated memory. If memory is constrained, WCF might be a viable option;

A note on the chart: I've tried my best to make it look good on desktop and on mobile, but the viewing experience is still best on a large screen.

METHOD	ITEMCOUNT	MEAN	
SmallWcfText	0	365,8 µs	383,1
SmallWcfWebXml	0	378,9 µs	392,0
SmallWcfWebJson	0	396,1 µs	421,4
SmallWebApiJsonNet	0	1 802,4 µs	1 904,6
SmallWebApiMessagePack	0	931,8 µs	955,5
SmallWebApiXml	0	997,7 µs	1 067,0
SmallWebApiUtf8Json	0	925,5 µs	939,4
SmallAspNetCoreJsonNet	0	1 753,1 µs	1 776,1
SmallAspNetCoreMessagePack	0	853,2 µs	898,7
SmallAspNetCoreXml	0	955,9 µs	1 014,9
SmallAspNetCoreUtf8Json	0	914,4 µs	957,6
LargeWcfText	0	353,1 µs	361,0
LargeWcfWebXml	0	374,8 µs	382,4
LargeWcfWebJson	0	386,9 µs	398,0

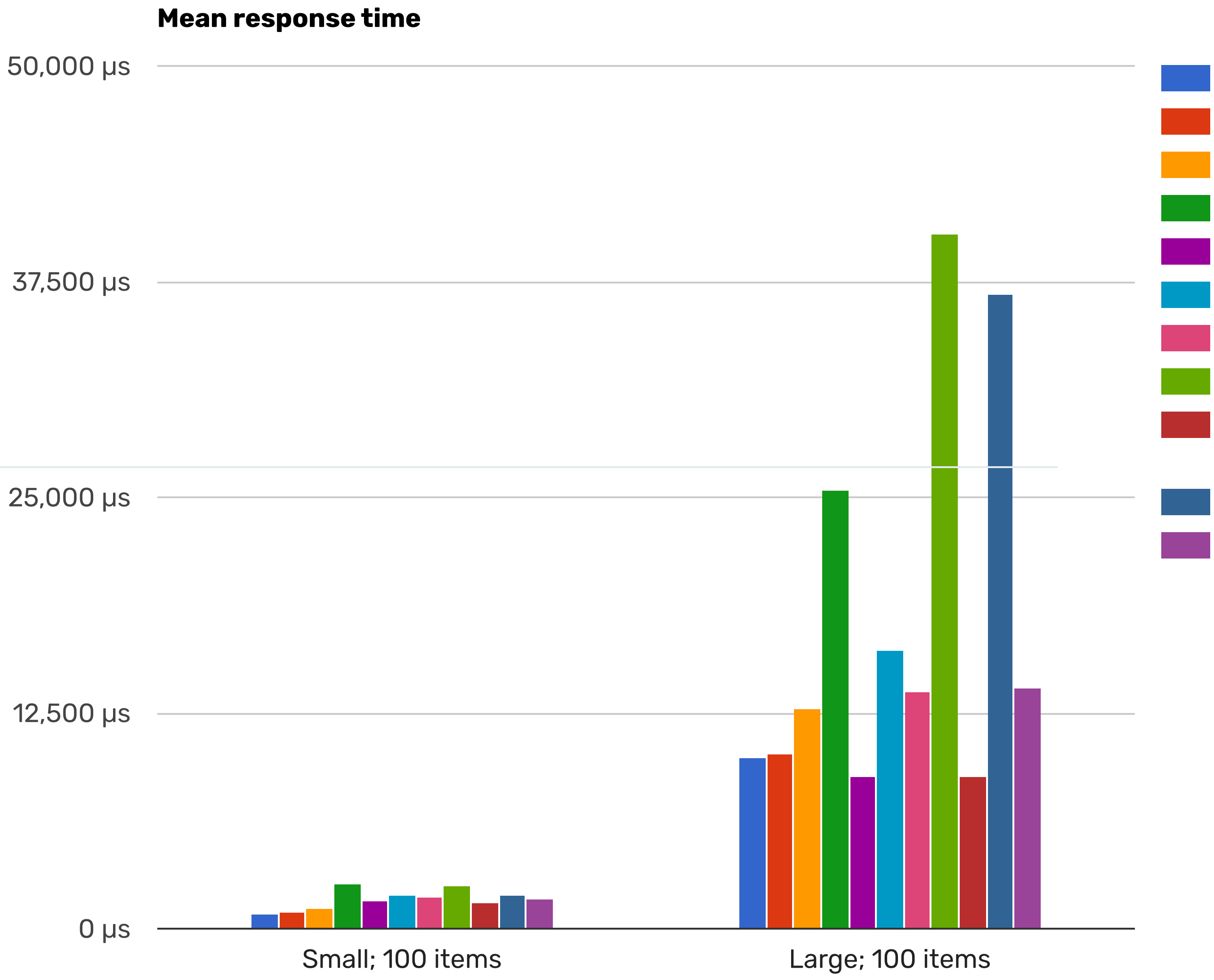
METHOD	ITEMCOUNT	MEAN	I
LargeWebApiJsonNet	0	1 860,5 μs	1 900,6
LargeWebApiMessagePack	0	924,2 μs	937,6
LargeWebApiXml	0	1 000,3 μs	1 110,8
LargeWebApiUtf8Json	0	929,3 μs	939,1
LargeAspNetCoreJsonNet	0	1 848,1 μs	1 915,2
LargeAspNetCoreMessagePack	0	881,4 μs	918,2
LargeAspNetCoreXml	0	943,4 μs	974,8
LargeAspNetCoreUtf8Json	0	943,7 μs	991,1
SmallWcfText	10	465,0 μs	534,4
SmallWcfWebXml	10	472,3 μs	516,5
SmallWcfWebJson	10	457,7 μs	467,2
SmallWebApiJsonNet	10	1 856,1 μs	1 890,8
SmallWebApiMessagePack	10	930,9 μs	940,6
SmallWebApiXml	10	1 267,1 μs	1 463,6
SmallWebApiUtf8Json	10	960,2 μs	1 027,8
SmallAspNetCoreJsonNet	10	1 899,6 μs	1 950,9
SmallAspNetCoreMessagePack	10	902,6 μs	933,2
SmallAspNetCoreXml	10	1 051,0 μs	1 186,4
SmallAspNetCoreUtf8Json	10	996,0 μs	1 057,8
LargeWcfText	10	1 471,8 μs	1 731,5
LargeWcfWebXml	10	1 430,8 μs	1 564,6

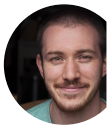
METHOD	ITEMCOUNT	MEAN	
LargeWcfWebJson	10	1 669,1 μs	1 822,5
LargeWebApiJsonNet	10	10 642,6 μs	11 767,8
LargeWebApiMessagePack	10	1 945,1 μs	2 036,7
LargeWebApiXml	10	2 487,6 μs	2 522,3
LargeWebApiUtf8Json	10	1 949,1 μs	2 098,9
LargeAspNetCoreJsonNet	10	10 367,4 μs	10 987,8
LargeAspNetCoreMessagePack	10	1 946,0 μs	1 975,3
LargeAspNetCoreXml	10	2 463,8 μs	2 521,2
LargeAspNetCoreUtf8Json	10	1 977,8 μs	2 084,4
SmallWcfText	100	830,1 μs	871,4
SmallWcfWebXml	100	961,7 μs	1 062,7
SmallWcfWebJson	100	1 188,1 μs	1 359,0
SmallWebApiJsonNet	100	2 614,2 μs	2 688,8
SmallWebApiMessagePack	100	1 615,7 μs	1 762,6
SmallWebApiXml	100	1 982,7 μs	2 020,6
SmallWebApiUtf8Json	100	1 816,0 μs	1 860,2
SmallAspNetCoreJsonNet	100	2 524,8 μs	2 608,3
SmallAspNetCoreMessagePack	100	1 483,8 μs	1 792,1
SmallAspNetCoreXml	100	1 933,5 μs	1 960,5
SmallAspNetCoreUtf8Json	100	1 777,2 μs	1 831,4
LargeWcfText	100	9 890,2 μs	10 865,7

METHOD	ITEMCOUNT	MEAN	
LargeWcfWebXml	100	10 136,1 µs	11 027,3
LargeWcfWebJson	100	12 813,6 µs	13 895,2
LargeWebApiJsonNet	100	25 425,6 µs	26 789,2
LargeWebApiMessagePack	100	8 834,9 µs	9 149,7
LargeWebApiXml	100	16 193,5 µs	16 481,3
LargeWebApiUtf8Json	100	13 787,9 µs	14 252,9
LargeAspNetCoreJsonNet	100	40 312,9 µs	43 191,0
LargeAspNetCoreMessagePack	100	8 813,8 µs	10 180,6
LargeAspNetCoreXml	100	36 767,1 µs	39 825,2
LargeAspNetCoreUtf8Json	100	13 961,1 µs	14 168,9
SmallWcfText	1000	5 299,2 µs	6 077,0
SmallWcfWebXml	1000	5 758,0 µs	7 026,5
SmallWcfWebJson	1000	6 747,8 µs	7 438,1
SmallWebApiJsonNet	1000	9 909,7 µs	10 125,5
SmallWebApiMessagePack	1000	4 187,4 µs	4 299,4
SmallWebApiXml	1000	9 301,4 µs	9 428,8
SmallWebApiUtf8Json	1000	4 849,8 µs	4 906,9
SmallAspNetCoreJsonNet	1000	18 942,1 µs	19 746,0
SmallAspNetCoreMessagePack	1000	3 682,6 µs	3 756,6
SmallAspNetCoreXml	1000	25 635,3 µs	26 039,3
SmallAspNetCoreUtf8Json	1000	4 896,0 µs	4 979,0

METHOD	ITEMCOUNT	MEAN	
LargeWcfText	1000	111 347,5 µs	113 444,9
LargeWcfWebXml	1000	110 655,4 µs	119 280,8
LargeWcfWebJson	1000	134 667,6 µs	144 168,1
LargeWebApiJsonNet	1000	173 466,7 µs	201 296,3
LargeWebApiMessagePack	1000	77 229,8 µs	79 016,2
LargeWebApiXml	1000	155 180,8 µs	156 540,0
LargeWebApiUtf8Json	1000	113 577,7 µs	115 954,0
LargeAspNetCoreJsonNet	1000	364 533,8 µs	366 468,4
LargeAspNetCoreMessagePack	1000	75 584,5 µs	76 337,7
LargeAspNetCoreXml	1000	355 358,8 µs	359 180,4
LargeAspNetCoreUtf8Json	1000	105 684,4 µs	107 125,5

100 items ▼ Mean response time ▼


Mean response time
50,000 μs
37,500 μs
25,000 μs
12,500 μs
0 μs
Small; 100 items
Large; 100 items



3 Comments **Erik Heemskerk** **Mehmet TEMEL**

♡ **Recommend** **Share** **Sort by Best**

Join the discussion…

Kralizek • a day ago

It would have been great to see how WCF behaves with a protobuf serializer :)

Also using NetTcpBinding would have been a nice way to go for speed ;)

Reply • Share ›

Erik Heemskerk Mod → Kralizek • a day ago

I decided against that because the benchmarks on the MessagePack GitHub repo already show it is faster than protobuf-net.

Also I didn’t want to include NetTcpBinding, because comparing, basically, a raw socket to an HTTP request is unfair.

Reply • Share ›

Vince Zalamea → Erik Heemskerk • 20 hours ago

We use a custom binding (HTTP transport with binary encoding). Pretty fast for our internal uses. It's pretty best-of-both-worlds: we're crossing machine boundaries so named pipes is out. Plus, since we're a Microsoft shop, we can use the proprietary binary encoding. Fast enough for our needs. We also add some Gzip compression for an extra boost.

Reply • Share ›

ALSO ON **ERIK HEEMSKERK**

C# 7.x and 8.0: Uncertainty and Awesomeness

13 comments • 9 months ago

RobIII — > "The C# team have, for now, chosen to do the opposite of marking a variable as non-nullable; all reference types

Event Sourcing: Querying using read models

6 comments • a year ago

Erik Heemskerk — You don’t necessarly need to have the read model stores on the instances themselves. If you anticipate that you need to

DDD Persistence - Record Event-Driven Persistence

23 comments • 4 months ago

Event Sourcing: Awesome, powerful & different

4 comments • a year ago

.NET CORE

PSA: Don’t change the assembly name for published NuGet packages

Somebody published a new version of a NuGet package with a different assembly name. You’ll never guess what happens next.

ERIK HEEMSKERK

Latest Posts Twitter Ghost Graphics by MargaHD