

Avro Api as evolution of JSON

Adrian Strugala

December 24, 2020

The article shows a way of evolution of standard JSON APIs based on the author's experience. Solution based on Avro format significantly reduces time and network traffic in communication between microservices. The article includes results of benchmark comparing JSON API with variations of Avro APIs and implementation details in C# .NET Core.

1 Introduction

Hi! I am a software developer working in C# .NET environment, focused mostly on the backend side of the applications. That means I am delivering the data. Fetching the data. Synchronizing the data. Downloading the data. Checking data quality. Pulling the data. Mixing together data from various sources to provide new data. I think you know what I am talking about.

Fortunately, I am living in a microservice world, where the data is well organized. The flag project of my company is build of 40-50 services which exposes about 500 endpoints in total. Even my side project is built of 6 services (20 APIs). I am using 3rd party APIs and open APIs. During this everyday job, I noticed how to improve the most popular JSON APIs.

Believe me or not, services love to talk to each other. They do this all the time and that's good. My customers are able to see the data, manipulate it and delete it. Background jobs are generating reports, documents or whatever they want. The problem starts, when the communication slows the services down and they are not able to do their job properly.

2 The problem

Some time ago developers in my company were asked to limit calls performed against on-premise microservices, as surprisingly the problem was the local internet bandwidth throughput. Local servers were connected by 100 Mb/s network and were unable to handle heavy traffic.

A few days later I heard a conversation between my colleague and his Product Owner. The PO asked If there is any quick-win on how to improve the response time of his service. The colleague started to explain what is the root cause of

the problem: his service was fetching data from API A, then B, C, D, and E. The final response time was strongly dependent on the connected services.

Then colleague, who is a great professional started to enumerate possible solutions: cache part of the data, go in the direction of CQRS and Event Sourcing - start pre-generating view models as soon as the data changes. His answers were right, but caching in live-APIs is sometimes impossible. Implementation of Event Sourcing is very, very expensive both in terms of implementation as well as changing developers' approach in the existing environment.

There is also an additional reason, why my company wants to reduce communication time and reduce storage costs. We are slowly moving to the Internet of Things and Big Data technologies. And, in fact, Big Data workshops was the place where I learned about Avro format.

I thought about those problems and I found one, simple solution fulfilling 3 main assumptions:

- Reduce the network traffic
- Decrease the communication time between microservices
- Low implementation cost

But, first things first. I will start with a few words about why we are all currently using JSON APIs.

3 JSON - as the standard

JSON format implements a number of key features, that I could not imagine using REST API without. The most important are: the clear and easily readable format and consistent data model. Also, it is worth mentioning the number of tools you can use to parse, read or edit JSONs and even generate it automatically from C# models.

In fact, JSON has only one main disadvantage that comes to my mind - every response and request is sent as plain text. Usually, it is not a big deal, but in the case under consideration lack of default compression mechanism was the factor that brought the topic to the table.

4 Avro - as the evolution

Let me now briefly introduce the Avro format. For the detailed description please follow links to the Apache Wiki at the end of the article.

Avro file is build of 2 main parts:

1. Header
2. Data

Header contains information about used codec (compression algorithm) and readable schema of the data.

The data itself is compressed to the binary representation. Take a look at the illustrative example:

JSON	Avro
<pre>[{ "Id": 208049411, "IsActive": true, "Name": "Name053b70e6-3935-46bf-9bbd-4e5e0a714084", "Age": 1431481648, "Address": "Addressc0d5ba1a-24f3-4538-a49f-42f70cf5d0b0" }, { "Id": 107017917, "IsActive": false, "Name": "Name5412ad55-b36a-41a2-94b9-7f6bdccaa649", "Age": 127848555, "Address": "Addressf039ea17-892b-44ee-8e56-29cd681ef40f" }, { "Id": 335432345, "IsActive": true, "Name": "Name4c08f8f7-b7ba-48ee-8993-893440135d8f", "Age": 1426027908, "Address": "Address9875dfcb-c2a2-4a27-9ee5-2577cf84b4ee" }, { "Id": 1794552263, "IsActive": false, "Name": "Name7aedbd1c-1e97-4b31-9434-feb0f55e851b", "Age": 1809071694, "Address": "Address6562d18e-22f1-449f-9a1d-e5aea74d09ea" }, { "Id": 1536567078, "IsActive": true, "Name": "Name4b3be4a4-493e-4dfc-978e-26d9ff63ade6", "Age": 1560932751, "Address": "Address243787aa-42da-4076-a18c-9da7ef901719" }, { "Id": 874460291, "IsActive": false, "Name": "Name813dcccfe-17e7-4bfd-ab6c-2a16832f018", "Age": 555225105, "Address": "Address3b5ca63b-9dc8-41ac-889d-157fbfe3ac46" }, { "Id": 1959451664, "IsActive": true, "Name": "Name2b92ee5e-ce28-425b-b269-997f5b772532", "Age": 1383547002, "Address": "Address18beaff6-5bdc-4e68-bc40-230c42d4431f" }]</pre>	<pre>Obj1:avro.codec:null:avro.schema: { "type": "array", "items": { "type": "record", "name": "User", "namespace": "SolTechnology.Demo", "fields": [{ "name": "Id", "type": "int" }, { "name": "IsActive", "type": "boolean" }, { "name": "Name", "type": "string" }, { "name": "Age", "type": "int" }, { "name": "Address", "type": "string" }] } }</pre>

Figure 1: Comparison of JSON and Avro formats

Json:

```
1 [
2   {
3     "Id": 208049411,
4     "IsActive": true,
5     "Name": "Name053b70e6-3935-46bf-9bbd-4e5e0a714084",
6     "Age": 1431481648,
7     "Address": "Addressc0d5ba1a-24f3-4538-a49f-42f70cf5d0b0"
8   },
9   {
10    "Id": 107017917,
11    "IsActive": false,
```

```

12      "Name": "Name5412ad55-b36a-41a2-94b9-7f8
        bdccaa649",
13      "Age": 727845855,
14      "Address": "Addressf039ea17-892b-44ee-8e56-2
        9cd681ef40f"
15    },
16    {
17      "Id": 335432365,
18      "IsActive": true,
19      "Name": "Namec4c8f8f7-b7ba-48ee-8993-8934401
        35d8f",
20      "Age": 1426027908,
21      "Address": "Address9875dfcb-c2a2-4a27-9ee5-2
        577cf86b4ee"
22    },
23    {
24      "Id": 1784552263,
25      "IsActive": false,
26      "Name": "Name7aebd1c-1e97-4b31-9434-feb0f55
        6851b",
27      "Age": 1809071694,
28      "Address": "Address6562d18e-22f1-449f-9a1d-e
        5aea74d09ea"
29    },
30    {
31      "Id": 1536567078,
32      "IsActive": true,
33      "Name": "Name483be4a4-493e-4dfc-978e-26d9ff6
        3ade6",
34      "Age": 1560932751,
35      "Address": "Address243787aa-42da-4076-a18c-9
        da7ef901719"
36    },
37    {
38      "Id": 874460291,
39      "IsActive": false,
40      "Name": "Name81ddccfe-17e7-4bfd-ab6c-2a16833
        2f018",
41      "Age": 555229108,
42      "Address": "Address3b5ca63b-9dc8-41ac-889d-1
        57fbfe3ac46"
43    },
44    {
45      "Id": 1958451664,
46      "IsActive": true,
47      "Name": "Name2b92ee5e-ce28-425b-b269-997f5b7

```

```

48         72532",
49         "Age": 1383547002,
50         "Address": "Address18bea6f6-5bdc-4e68-bc40-2
51         30c42d4431f"
52     }
53 ]

```

Avro:

```

1  Obj1;avro.codec:null;avro.schema:
2  {
3      "type": "array",
4      "items": {
5          "type": "record",
6          "name": "User",
7          "namespace": "SolTechnology.Demo",
8          "fields": [
9              {
10                 "name": "Id",
11                 "type": "int"
12             },
13             {
14                 "name": "IsActive",
15                 "type": "boolean"
16             },
17             {
18                 "name": "Name",
19                 "type": "string"
20             },
21             {
22                 "name": "Age",
23                 "type": "int"
24             },
25             {
26                 "name": "Address",
27                 "type": "string"
28             }
29         ]
30     }
31 }
32 54f05db088f44c9bb6a2d64bc6c7d43e14e0d2f8260c42b2af5e
33 58d3d3bd1712
34 cc3c4bbece6543ed99f81e5a1ceaad11728cc8d80c4b45d7869a
35 0a6055e68e83
36 8612ed73466a4bfdab8c5832cbc2153f94283a270a9a45229af5
37 c021d9abbf78

```

```

35 960454641f104e59beadff09d36580ede75441d74d064d18a980
    3fa304b1e652
36 923695eb37654b66ad93af65eb38339f2be446600cb344f5b5bc
    1ec1f182adb1
37 9f618c344e71443dba8c43578143db870167055a569346d3b0c3
    ac6b5f98cce3
38 1fb631ff460044d8b48008f1dcecd62e5f62e94008bf467e8bc1
    73d38b213749

```

It doesn't look really impressive here. But imagine a very, very long Json. The size of the file would increase linearly with the number of records. While for Avro header and schema stays the same - what increases is the amount of encoded and well-compressed data.

Avro format inherits the readability of Json. Note the schema representation - it could be easily read and extracted from the content. In real-life cases, this is very helpful e.g. during integration tests. I can call an API, and read just the schema of the data model - to prepare my classes for deserialization.

Take a look at the data - you are not able to read it at first glance. And that is also a benefit. API responses could be easily hijacked by network tools. You can even peek the responses in internet browsers. And from time to time it happens, that someone spots the data that shouldn't be read by an unauthorized person. Keeping the data encoded increases the security of the solution. Reading Avro is not a big problem for a motivated person, but reduces the probability of accidental data leaks.

5 The benchmark results

Moving to raw data, numbers. The table below shows results of BenchmarkDotNet performing request againsts Json and Avro APIs sending **the same response** but configured with different response serializers.

BenchmarkDotNet=v0.12.1, OS=Windows 10.0.18363.1139 (1909/November2018Update/19H2) Intel Core i7-7820HQ CPU 2.90GHz (Kaby Lake), 1 CPU, 8 logical and 4 physical cores .NET Core SDK=3.1.402 [Host] : .NET Core 3.1.8 (CoreCLR 4.700.20.41105, CoreFX 4.700.20.41903), X64 RyuJIT [AttachedDebugger] DefaultJob : .NET Core 3.1.8 (CoreCLR 4.700.20.41105, CoreFX 4.700.20.41903), X64 RyuJIT

Serializer	Request Time	Serializer Time*	Allocated Memory	Compressed Size
Json	672.3 ms	173.8 ms	52.23 MB	6044 kB
Avro	384.7 ms	159.2 ms	76.58 MB	2623 kB
JsonGzip	264.1 ms	232.6 ms	88.32 MB	514 kB
JsonBrotli	222.5 ms	210.5 ms	86.15 MB	31 kB
AvroBrotli	193.5 ms	184.7 ms	74.75 MB	31 kB
AvroGzip	181.2 ms	168.5 ms	75.05 MB	104 kB

*Time needed only for serialization and deserialization

One of the greatest features of Avro format is the possibility to choose codec type (compression algorithm in fact) used for serialization of the data. In these examples, GZip and Deflate encodings are clear winners. Enabling one of them decreases file size to only 200 KB - this means: 50 time less than Json.

For the purpose of comparison, I have included also gzipped Json file. This solution greatly decrease the file size but deprive all of the advantages of Json format (readability etc.)







Name	Date modified	Type	Size
 10mega.json	31.03.2020 09:30	JSON File	9 945 KB
 10mega.avro	31.03.2020 09:30	AVRO File	2 436 KB
 10mega.snappy.avro	31.03.2020 09:30	AVRO File	421 KB
 10mega.json.gz	31.03.2020 09:30	GZ File	257 KB
 10mega.GZip.avro	31.03.2020 09:30	AVRO File	206 KB
 10mega.deflate.avro	31.03.2020 09:30	AVRO File	206 KB

Figure 2: Comparison of the same file compressed with different options

At this point let me introduce a library that I have created in the purpose of handling serialization and deserialization C# objects to Avro format: AvroConvert: Github page. I have strongly focused on the dev workflow and usability of the package. Its interface should be clear and familiar for every user.

Benchmark below mimics sending API response. Calculates the time of serialization of the message, time for used for transport (based on size) and time of deserialization:

```

AvroConvert Benchmark - fire!

The Benchmark compares Newtonsoft.Json, Apache.Avro, AvroConvert (nuget version)
and AvroConvert local version
Number of runs: 50
Progress: 10/50
Progress: 20/50
Progress: 30/50
Progress: 40/50
Progress: 50/50

Json:          Serialize: 96 ms 11138 kB; Deserialize: 138 ms
Apache.Avro:   Serialize: 133 ms 6283 kB; Deserialize: 168 ms
Avro Headless: Serialize: 85 ms 6283 kB; Deserialize: 202 ms
Avro Deflate:  Serialize: 273 ms 3080 kB; Deserialize: 246 ms
Avro vNext:    Serialize: 82 ms 6283 kB; Deserialize: 200 ms

Summarize:
Let's produce one value for each of the records. Assume internet speed 100mb/s
and calculate how fast the data is send between to microservices:

Json:          1104,2091217041016 ms
Apache.Avro:   791,9137725830078 ms
Avro Headless: 777,9140777587891 ms
Avro Deflate:  759,6292724609375 ms
Avro vNext:    772,9140777587891 ms

```

Figure 3: Comparison of result from AvroConvert benchmark

All of the implementations of Avro provides quite similar results. The reason, why I didn't choose the official implementation of Avro format in the C# world (Apache.Avro) is that it doesn't support many key types like decimal, dictionary, list or Guid.

AvroConvert supports few serialization scenarios:

1. Standard serialization - result contains header and data. Like in the given example.
2. Serialization with additional encoding - result contains a header (with information about codec type) and encoded data. It decreases the size of the result but increases serialization and deserialization time. Look at the AvroConvert Deflate on the benchmark screen.
3. Headless serialization - result contains only data. Assumes, that schema is known upfront. Decreases size of the result, serialization and deserialization time.

To sum up - in the given scenario it doesn't really matter which Avro serialization you will choose. Each of them speed's up the response for about 25-30%.

6 How to build Avro API

And finally - let's code this! The implementation in .NET Core 3.0 is very easy, in fact, we need just 3 classes:

- AvroInputFormatter
- AvroOutputFormatter
- HttpClient extensions that support Avro format

In my implementation serializaiton is done by AvroConvert. Every implementation of Avro serializaiton should be compatible with each other, though.

6.1 AvroInputFormatter

```
1 public class AvroInputFormatter : InputFormatter
2 {
3     public AvroInputFormatter()
4     {
5         this.SupportedMediaTypes.Clear();
6
7         this.SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("
            application/avro"));
8     }
9
10    public override Task<InputFormatterResult>
        ReadRequestBodyAsync(InputFormatterContext context)
11    {
12        using (MemoryStream ms = new MemoryStream())
13        {
14            context.HttpContext.Request.Body.CopyTo(ms);
15            var type = context.ModelType;
16
17            object result = AvroConvert.Deserialize(ms.ToArray(), type
                );
18            return InputFormatterResult.SuccessAsync(result);
19        }
20    }
21 }
```

6.2 AvroOutputFormatter

```
1 public class AvroOutputFormatter : OutputFormatter
2 {
3     public AvroOutputFormatter()
4     {
5         this.SupportedMediaTypes.Clear();
6
7         this.SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("
            application/avro"));
8     }
9 }
```

```

9
10 public override async Task WriteResponseBodyAsync(
    OutputFormatterWriteContext context)
11 {
12     var avroBody = AvroConvert.Serialize(context.Object);
13
14     var response = context.HttpContext.Response;
15     response.ContentLength = avroBody.Length;
16
17     await response.Body.WriteAsync(avroBody);
18 }
19 }

```

6.3 HttpClient extensions

```

1 public static class HttpClientExtensions
2 {
3     public static async Task<HttpResponseMessage> PostAsAvro(this
        HttpClient httpClient, string requestUri, object content)
4     {
5         var body = new ByteArrayContent(AvroConvert.Serialize(
            content));
6         body.Headers.ContentType = new MediaTypeHeaderValue("
            application/avro");
7         return await httpClient.PostAsync(requestUri, body);
8     }
9
10    public static async Task<T> GetAsAvro<T>(this HttpClient
        httpClient, string requestUri)
11    {
12        var response = await httpClient.GetByteArrayAsync(requestUri
            );
13        T result = AvroConvert.Deserialize<T>(response);
14        return result;
15    }
16 }

```

6.4 Modify Startup

```

1 services.AddMvc(options =>
2 {
3     options.InputFormatters.Insert(0, new AvroInputFormatter());
4     options.OutputFormatters.Insert(0, new AvroOutputFormatter());
5 });

```

And - that's it. You've just speeded up responses of your APIs by at least 30%. Play with the serialization options, you can achieve even better results. I've gathered the methods used for communication in the separate library: SolTechnology.Avro.Http.

Thank you for reading the article. I hope you can make good use of the knowledge I just shared. In case of any questions, contact me at strugala.adrian@gmail.com. See you!

7 Useful links

- <http://avro.apache.org/>
- <https://cwiki.apache.org/confluence/display/AVRO/Index>
- <https://github.com/AdrianStrugala/AvroConvert>