Avro Api as evolution of JSON

Adrian Strugala

December 27, 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

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

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

Figure 1: Comparison of JSON and Avro formats

Json:

```
[
    {
        "Id": 208049411,
        "IsActive": true,
        "Name": "Name053b70e6-3935-46bf-9bbd-4e5e0a714
            084",
        "Age": 1431481648,
        "Address": "Addressc0d5ba1a-24f3-4538-a49f-42f
            70cf5d0b0"
    },
    {
        "Id": 107017917,
        "IsActive": false,
```

```
        "Name": "Name5412ad55-b36a-41a2-94b9-7f8bdccaa649",
        "Age": 727845855,
        "Address": "Addressf039ea17-892b-44ee-8e56-29cd681ef40f"
    },
    {
        "Id": 335432365,
        "IsActive": true,
        "Name": "Namec4c8f8f7-b7ba-48ee-8993-893440135d8f",
        "Age": 1426027908,
        "Address": "Address9875dfcb-c2a2-4a27-9ee5-2577cf86b4ee"
    },
    {
        "Id": 1784552263,
        "IsActive": false,
        "Name": "Name7aedbd1c-1e97-4b31-9434-feb0f556851b",
        "Age": 1809071694,
        "Address": "Address6562d18e-22f1-449f-9a1d-e5aea74d09ea"
    },
    {
        "Id": 1536567078,
        "IsActive": true,
        "Name": "Name483be4a4-493e-4dfc-978e-26d9ff63ade6",
        "Age": 1560932751,
        "Address": "Address243787aa-42da-4076-a18c-9da7ef901719"
    },
    {
        "Id": 874460291,
        "IsActive": false,
        "Name": "Name81ddccfe-17e7-4bfd-ab6c-2a168332f018",
        "Age": 555229108,
        "Address": "Address3b5ca63b-9dc8-41ac-889d-157fbfe3ac46"
    },
    {
        "Id": 1958451664,
        "IsActive": true,
        "Name": "Name2b92ee5e-ce28-425b-b269-997f5b772
```

```
            532",
        "Age": 1383547002,
        "Address": "Address18bea6f6-5bdc-4e68-bc40-230
            c42d4431f"
    }
]
```

Avro:

```
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"
            }
        ]
    }
}
54f05db088f44c9bb6a2d64bc6c7d43e14e0d2f8260c42b2af5e58
    d3d3bd1712
cc3c4bbece6543ed99f81e5a1ceaad11728cc8d80c4b45d7869a0a
    6055e68e83
8612ed73466a4bfdab8c5832cbc2153f94283a270a9a45229af5c0
    21d9abbf78
```

```
35 960454641f104e59beadff09d36580ede75441d74d064d18a9803
       fa304b1e652
36 923695eb37654b66ad93af65eb38339f2be446600cb344f5b5bc1
       ec1f182adb1
37 9f618c344e71443dba8c43578143db870167055a569346d3b0c3ac
       6b5f98cce3
38 1fb631ff460044d8b48008f1dcecd62e5f62e94008bf467e8bc173
       d38b213749
```

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 models 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 at the responses in internet browsers. And from time to time it happens, that someone spots the data that should not 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 numbers. The table below shows results of BenchmarkDotNet performing request against 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

Serializers used in this experiment are Newtonsoft.Json and AvroConvert, a library that I have created for 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.

In the benchmark, standard JSON API provides the slowest response, and the returned object is of the biggest size. Just switching the serialization library to AvroConvert reduced response size by 56.6% and time by 42.8%(!!!). After setting up the proper codec for this case the final result was: **3 times the request time reduction and result object 60 times smaller than initial.**

If we are working in modern infrastructure, and the network is not a very big deal, we can take a look only at serializer library time. Avro library performs about 15 ms faster than JSON. Maybe not much, but scale matters. When an API response 100 times per day it is 1.5 s gain, if 1000 times, 15 s. If this API responses every day: the gain is 1.5 hours per year.

For the comparison purpose, I have included also JSON API with data compressed with GZip and Brotli compressions. The results are quite nice, but deprive all of the advantages of JSON format - the whole response is compressed and not readable anymore and the implementation is a little bit more complex than it was before. The real problem starts when my microservice calls a few others and they are returning responses in different compression formats. I would have to check manually the formats and write classes handling each of them. Not very convenient nor automated.

Using Avro, this problem disappears. Avro deserializer discovers used codec by itself and automatically deserializes the data. No difference if other API responses with gzipped, deflated, or raw Avro result.

What is more, 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. When the size is the key factor, for example when the case is to store the amount of data, the codec can be changed just by selecting a different enum value during serialization. In the example above, Brotli would be a clear winner. Enabling it decreases object size to only 31 kB - this means: 200 times less than JSON.

To sum up - in the given scenario Avro API fulfills every of the given assumptions:

- Network traffic reduced by 98%
- Communication time between microservices decreased by 73%
- Implementation is simple - look at the section below

6 How to build Avro API

And finally - let's code this! The implementation in .NET Core 3.1 is very easy, in fact, we need just 3 classes:

- AvroInputFormatter
- AvroOutputFormatter
- HttpClient extensions that support Avro format

In my implementation serialization is done by AvroConvert. Every implementation of Avro serialization 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 async Task<InputFormatterResult>
           ReadRequestBodyAsync(InputFormatterContext context)
11     {
12         await using MemoryStream ms = new MemoryStream();
13         await context.HttpContext.Request.Body.CopyToAsync(ms);
14         var type = context.ModelType;
15
16         object result = AvroConvert.Deserialize(ms.ToArray(), type);
17         return await InputFormatterResult.SuccessAsync(result);
18     }
19 }
```

6.2 AvroOutputFormatter

```
1  public class AvroOutputFormatter : OutputFormatter
2  {
3      private readonly CodecType _codec;
4
5      public AvroOutputFormatter(CodecType codec = CodecType.Null)
6      {
7          _codec = codec;
8          this.SupportedMediaTypes.Clear();
9
10         this.SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("
               application/avro"));
11     }
12
13     public override async Task WriteResponseBodyAsync(
           OutputFormatterWriteContext context)
14     {
15         var avroBody = AvroConvert.Serialize(context.Object, _codec)
               ;
```

```
16
17          var response = context.HttpContext.Response;
18          response.ContentLength = avroBody.Length;
19
20          await response.Body.WriteAsync(avroBody);
21      }
22  }
```

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. Have a nice day!

7 Useful links

- http://avro.apache.org/

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