Why Avro API is the best choice?

Adrian Strugala

March 30, 2020

In the first section, I will briefly introduce myself. Then I will describe the problem resolved in the article and compare Json and Avro formats. Finally, I will present the results of my benchmark and provide implementation details.

1 Introduction

Hi! I am a software developer working in C# .NET environment. I am 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 kindly asked to limit calls performed against on-premise microservices, as surprisingly the problem was the local internet bandwidth throughput. Funny or not, the solution from the management was really to reduce the traffic between microservices.

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.

I thought about those problems and I found out one, really simple solution that can bring 3 main benefits:

- Decrease the communication time between microservices

- Reduce the network traffic

- Increase communication security

But, first things first. I'll start with a few words about why we are all currently using Json APIs.

3   Why Json is amazing

That's simple - just try to imagine communication without Json. What would you miss the most? The clear and easily readable format? Consistent data model? Maybe the number of tools you can use to parse, read or edit Jsons and even generate it automatically from C# models?

If fact Json has only one main disadvantage that comes to my mind - every response and request is sent as plain text. Sometimes it's not a big deal, but in other cases response time of not compressesd nor encoded Json API could be a real problem.

4   Why Avro is better

Avro file is build of few pieces:

1. Magic number

2. Chosen codec (null in example)

3. Schema of the data written in Json format

4. The data itself compressed to binary representation

An example of exactly the same data:
Json:

```
1  [
2    {
3      "minPosition": 188,
4      "hasMoreItems": true,
```

```
 5      "itemsHtml": "items_html6e64c2b9-dc87-4be3-b8ba-
            eca0da96ce78",
 6      "newLatentCount": 85,
 7      "itemIds": [
 8      174,
 9      43,
10      249
11      ],
12      "isAvailable": false
13    },
14    {
15      "minPosition": 160,
16      "hasMoreItems": true,
17      "itemsHtml": "items_htmlaa233d3b-d6ea-41ff-b50f-
            f099c0c79991",
18      "newLatentCount": 163,
19      "itemIds": [
20      60,
21      153,
22      131
23      ],
24      "isAvailable": false
25    }
26  ]
```

Avro:

```
 1  Objavro.codecnullavro.schemaÂ¨{"type":"array","items
       ":{"type":"record","name":"Dataset","fields":[{"
       name":"minPosition","type":"int"},{"name":"
       hasMoreItems","type":"boolean"},{"name":"
       itemsHtml","type":["null","string"]},{"name":"
       newLatentCount","type":"int"},{"name":"itemIds",
       "type":{"type":"array","items":"int"}},{"name":"
       isAvailable","type":"boolean"}]}} /ĂĹşÄĬ)|
       Ĺ™Â OÄŚÃHE Ĺ™Ĺ™\items_html6e64c2b9-dc87-4be3-b8
       ba-eca0da96ce78ĹžĂśVĹ?  Ĺ"\items_htmlaa233d3b-d6
       ea-41ff-b50f-f099c0c79991Ä†xЀ  /ĂĹşÄĬ)|
       Ĺ™Â OÄŚÃHE
```

It doesn't look really different 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.
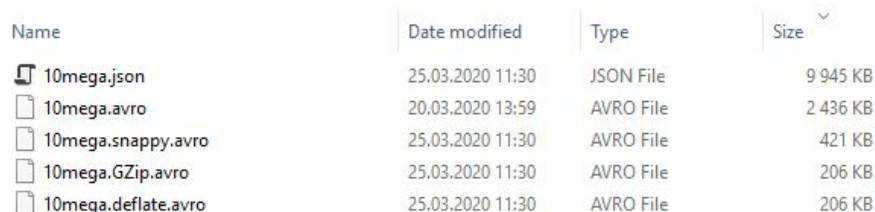
Take a look at the data - you are not able to read it at first glance. And that is also a benefit. API responses could be easily hijacked by network tools. You can even peek the responses in internet browsers. And from time to time it happens, that someone spots the data that shouldn't be read by an 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.

In a real-world case API responses are usually a little bit more complex than in this example. How beneficial is serialization using Avro in comparison to Json? Two times for this example. Three times for simple API responses. I was able to reach fifty(!) times using the right codec for nested model structures containing a huge amount of data - look at the screen below.

5   My benchmark results

Enough talking, lets now focus on numbers. The screen below shows data compressed with different formats and encodings. Take a careful look at the sizes. Json file, which size is about 10 000 KB was compressed to Avro file occupying about 2 500 KB (that's 4 times smaller).

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.

Name	Date modified	Type	Size
10mega.json	25.03.2020 11:30	JSON File	9 945 KB
10mega.avro	20.03.2020 13:59	AVRO File	2 436 KB
10mega.snappy.avro	25.03.2020 11:30	AVRO File	421 KB
10mega.GZip.avro	25.03.2020 11:30	AVRO File	206 KB
10mega.deflate.avro	25.03.2020 11:30	AVRO File	206 KB

Figure 1: Comparison of the same file compressed with different options

At this point let me introduce a library that I've created in the purpose of handling serialization and deserialization C# objects to Avro format: Avro-Convert: Github page. I've 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:

4

```
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 2: Comparison of result from AvroConvert benchmark

Winner in this category is the official implementation of Avro format in the C# world: Apache.Avro. Unfortunately, it doesn't support many key types like decimal, dictionary, list or Guid. That was the main reason why I decided to create AvroConvert.

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
```

```
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

- https://github.com/AdrianStrugala/SolTechnology.Avro.Http