Why Avro API is the best choice?

Adrian Strugala

March 28, 2020

1 Intruduction

Hi! I am a software developer working in C# .NET environment. I'm 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 produce new data. I think you know what I am talking about.

Fortunately, I am living in a microservice world. The data is well organized. The flag project of my company is build of 40-50 services. Each of them exposes from 5 up to 100 API endpoints. Even my side project is build of 6 services, 20 endpoints in total. I am using 3rd party APIs, public APIs, and open APIs. I mean - I know how to communicate between microservices. I do this every day.

Believe me or not, services love to talk to each other. They do this without any break. All the time. That's good. My customers are able to see the data, manipulate it and delete it. Background jobs are generating reports, documents and whatever they want. The problem starts, when the communication slows down the services and they are not able to play their role correctly.

2 The problem

Some time ago developers in my company were kindly asked to try to not call on-premise microservices more than it's needed. Surprisingly problem was the local internet bandwidth throughput. Funny or not, the solution from management was really to reduce 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 response time of his service. It wasn't that bad - just a little bit to slow for the users. The colleague started to explain what's the root cause of the problem: his service was fetching data from one API, then another, then 3rd one, authorizing and validating in the meantime. That means service A response time was strongly dependent on services B, C, D, and E. Then colleague as a great professionalist 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 in the existing environment.

I thought about those problems and I found out one, really simple solution which bringed 3 main benefits:

- Decrease the microservices communication time
- Reduce the network traffic
- Increase security between microservices

First things first, though. I'll start with a few words about why we are all in love with Json.

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 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,
```

```
      "itemIds": [
      174,
      43,
      249
      ],
      "isAvailable": false
    },
    {
      "minPosition": 160,
      "hasMoreItems": true,
      "itemsHtml": "items_htmlaa233d3b-d6ea-41ff-b50f-
         f099c0c79991",
      "newLatentCount": 163,
      "itemIds": [
      60,
      153,
      131
      ],
      "isAvailable": false
    }
]
```

Avro:

```
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 OÄŚĂHE Ĺ™Ĺ™\items_html6e64c2b9-dc87-4be3-b8
    ba-eca0da96ce78ĹžĂśVĹ?  Ĺ"\items_htmlaa233d3b-d6
    ea-41ff-b50f-f099c0c79991Ä†xË â€     /ĂĹşÄĬ)|
    Ĺ™Â OÄŚĂHE
```

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

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

Take a look at the data - you are not able to read it at first glace. And

that's also a benefit. API responses could be easly catched by network tools. You can even peek the responses in internet browsers. And from time to time happens, that someone spots the data that shouldn't be read by unauthorized person. Keeping data encoded increases security of the solution. Reading Avro is not a big problem for motivated person, but reduces probability of accidential 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? 2 times for this example. 3 times for simple API responses. I was able to reach 50 times using right codec for nested model structures containing huge amount of data.

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.

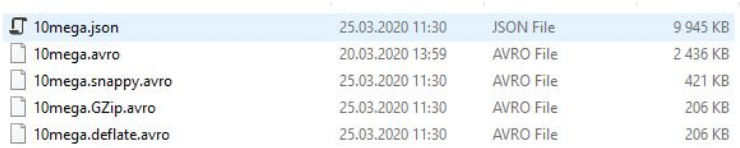

Figure 1: Comparison of the same file compressed with different options

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:

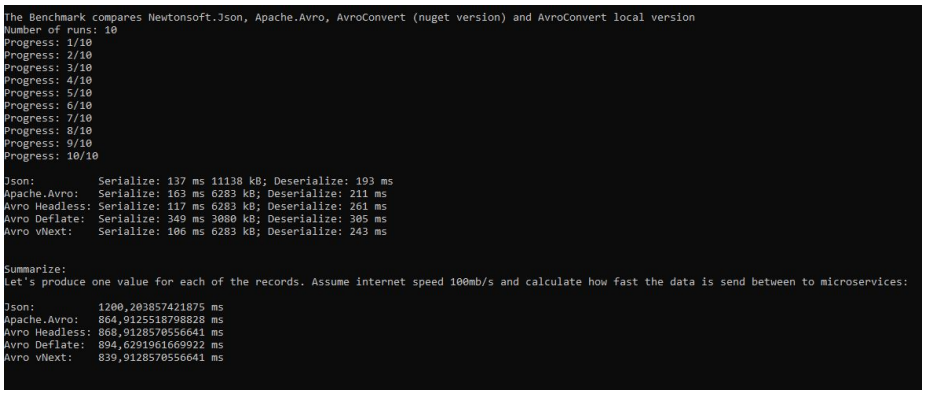

Figure 2: Comparison of result from AvroConvert benchmark

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

```

public class AvroInputFormatter : InputFormatter
{
  public AvroInputFormatter()
  {
    this.SupportedMediaTypes.Clear();

    this.SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("
        application/avro"));
  }

  public override Task<InputFormatterResult>
      ReadRequestBodyAsync(InputFormatterContext context)
  {
    using (MemoryStream ms = new MemoryStream())
    {
      context.HttpContext.Request.Body.CopyTo(ms);
      var type = context.ModelType;

      object result = AvroConvert.Deserialize(ms.ToArray(), type
          );
      return InputFormatterResult.SuccessAsync(result);
    }
  }
}
```

6.2 AvroOutputFormatter

```


public class AvroOutputFormatter : OutputFormatter
  {
  public AvroOutputFormatter()
  {
    this.SupportedMediaTypes.Clear();

    this.SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("
        application/avro"));
  }

  public override async Task WriteResponseBodyAsync(
      OutputFormatterWriteContext context)
  {
    var avroBody = AvroConvert.Serialize(context.Object);

    var response = context.HttpContext.Response;
    response.ContentLength = avroBody.Length;

    await response.Body.WriteAsync(avroBody);
  }
}
```

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