.NET 6 System.Text.Json

Ренат Тазиев (OZON Tech)

What's new?

- Source generators
- Writeble DOM
- IAsyncEnumerable serialization

Source generators: intro

- Improve the performance of application
  - move computing of serialization metadata from runtime to compile-time;
  - increased serialization throughput;
  - reduced start-up time;
  - reduced private memory usage;
  - removed runtime use of System.Reflection and System.Reflection.Emit;
  - trim-compatible serialization which reduces application size

Source generators: mode

- Serialization
  - generating source code that uses `Utf8JsonWriter` directly
  - only available for serialization
- Metadata
  - provides a static data access model
  - useful if you need reference handling and async serialization
  - provide benefits for deserialization
- Default
  - "everything on" mode

Source generators: how to use

- For old platform need install two packages:
  - Microsoft.CodeAnalysis
  - Microsoft.Net.Compilers.Toolset
- System.Text.Json supports by:
  - .NETCoreApp 6.0
  - .NETCoreApp 3.1
  - .NETStandard 2.0
  - .NETFramework 4.6.1

Source generators: how to use

```
// Data transfer model
public class DeliveryVariant
{
    public long Id { get; set; }
    public DeliveryVariantType Type { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
}
// Specified type for source generators
[JsonSerializable(typeof(DeliveryVariant))]
internal partial class MyJsonContext : JsonSerializerContext
{
}
// MyJsonContext is a partial class with the following shape
// internal partial class MyJsonContext : JsonSerializerContext
// {
//     public static MyJsonContext Default { get; }
//     public JsonTypeInfo<DeliveryVariant> DeliveryVariant { get; }
//     public MyJsonContext(JsonSerializerOptions options) { }
//     public override JsonTypeInfo GetTypeInfo(Type type) => ...;
// }
// Usage
var model = new DeliveryVariant {...};
var utf8Json = JsonSerializer.SerializeToUtf8Bytes(model, MyJsonContext.Default.DeliveryVariant);
model = JsonSerializer.Deserialize(utf8Json, MyJsonContext.Default.DeliveryVariant);
```

Source generators: how to use

```
builder.Services
    .AddControllers()
    .AddJsonOptions(options =>
    {
        options.JsonSerializerOptions.AddContext<MyJsonContext>();
    });
```

Source generators: how to use

```
<PropertyGroup>
    <EmitCompilerGeneratedFiles>true</EmitCompilerGeneratedFiles>
    <CompilerGeneratedFilesOutputPath>Generated</CompilerGeneratedFilesOutputPath>
</PropertyGroup>

<ItemGroup>
    <!-- Exclude the output of source generators from the compilation -->
    <Compile Remove="$(CompilerGeneratedFilesOutputPath)/**/*.g.cs" />
</ItemGroup>
```

Source generators: benchmarking

Method	Mean	StdDev	Ratio	Allocated
ClassicSerializer	313.7 us	4.55 us	1.00	154 KB
GeneratedSerializer	229.5 us	2.09 us	0.73	128 KB
ClassicSerializer_AsString	474.6 us	22.59 us	1.00	280 KB
GeneratedSerializer_AsString	277.0 us	18.81 us	0.58	254 KB
ClassicDeserializer	591.4 us	3.82 us	1.00	56 KB
GenerateDeserializer	467.5 us	3.55 us	0.79	56 KB

Source generators: mode

	JsonSerializer	JsonSerializer + pre-generating optimized serialization logic	JsonSerializer + pre-generating data access model
Simpler to code and debug	✔	✘	✘
Increases serialization throughput	✘	✔	✘
Reduces start-up time	✘	✔	✔
Reduces private memory usage	✘	✔	✔
Eliminates runtime reflection	✘	✔	✔
Facilitates trim-safe app size reduction	✘	✔	✔
Supports all serialization features	✔	✘	✔

Source generators: options

- JsonSourceGenerationOptions
- Supports:
  - DefaultIgnoreCondition
  - PropertyNamingPolicy
  - IncludeFields
  - IgnoreReadOnlyProperties
  - IgnoreReadOnlyFields
  - WriteIndented
- Unsupported:
  - DictionaryKeyPolicy
  - Encoder
  - IgnoreNullValues
  - NumberHandling
  - ReferenceHandler

Source generators: attributes

- Supports:
  - JsonIgnoreAttribute
  - JsonIncludeAttribute
  - JsonPropertyNameAttribute
- Unsupported:
  - JsonConverterAttribute
  - JsonExtensionDataAttribute
  - JsonNumberHandlingAttribute

Source generators: measuring

- K6 + local deployed service
- Measuring throughput
- 16 virtual user quering by 90 seconds

Source generators: measuring

```
                    : Standard | Serialization | Metadata
checks............: 100.00%   | 100.00%        | 100.00%
data_received.....: 12 GB     | 12 GB          | 12 GB
data_sent.........: 5.5 MB    | 5.6 MB         | 5.6 MB
avg...............: 387.69ms  | 380.38ms       | 378.92ms
min...............: 94.86ms   | 86.48ms        | 84.86ms
max...............: 1.15s     | 883.21ms       | 965.07ms
p(50).............: 377.55ms  | 373.56ms       | 369.5ms
p(90).............: 562.92ms  | 534.9ms        | 539.53ms
p(95).............: 622.76ms  | 589.92ms       | 591.47ms
http_reqs.........: 3717      | 3789           | 3804
throughput........: 40.86/s   | 41.69/s        | 41.89/s
```

Writable DOM: intro

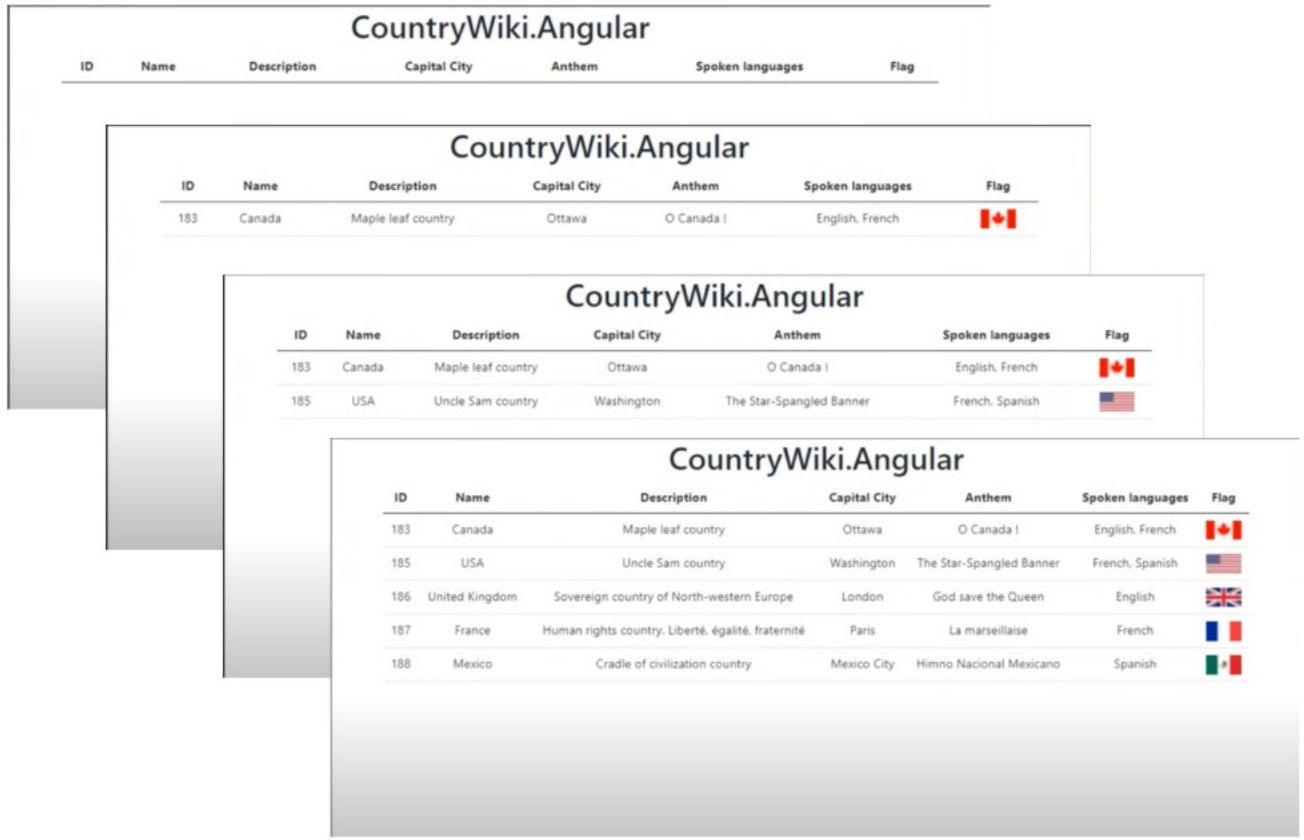
- A faster way to serialize objects when JSON Schema is not fixed
- Large DOM subsections can be modified faster and efficiently

Writable DOM: classes

Class/Type	Description
JsonDocument	Provides a mechanism for examining the structural content of a JSON value without automatically instantiating data values.
JsonNode	Abstract class Used To Parse Json string and Get Values from node.
JsonObject	Used to create Json object and can include JsonArray and Json Value
JsonArray	Used To Create JsonArray inside JsonObject
JsonValue	Abstract class, when obtained any value from JsonNode this type is returned using Dictionary Format.

IAsyncEnumerable serialization: intro

- Returning large JSON files without consuming a lot of memory
- Async loading of results



IAsyncEnumerable serialization: sample

```
[HttpGet(Name = "GetWeatherForecast")]
public IAsyncEnumerable<WeatherForecast> Get()
{
    return streamWeatherForecastsAsync();

    async IAsyncEnumerable<WeatherForecast> streamWeatherForecastsAsync()
    {
        var weatherForecast = Enumerable
            .Range(1, 15)
            .Select(index => new WeatherForecast
            {
                Date = DateTime.Now.AddDays(index),
                TemperatureC = Random.Shared.Next(-20, 55),
                Summary = Summaries[Random.Shared.Next(Summaries.Length)]
            })
            .ToArray();
        foreach (var forecast in weatherForecast)
        {
            await Task.Delay(Random.Shared.Next(150, 500));
            yield return forecast;
        };
    };
}
```

Thanks for watching

Improve the performance of your applications
Don't understand measure your results