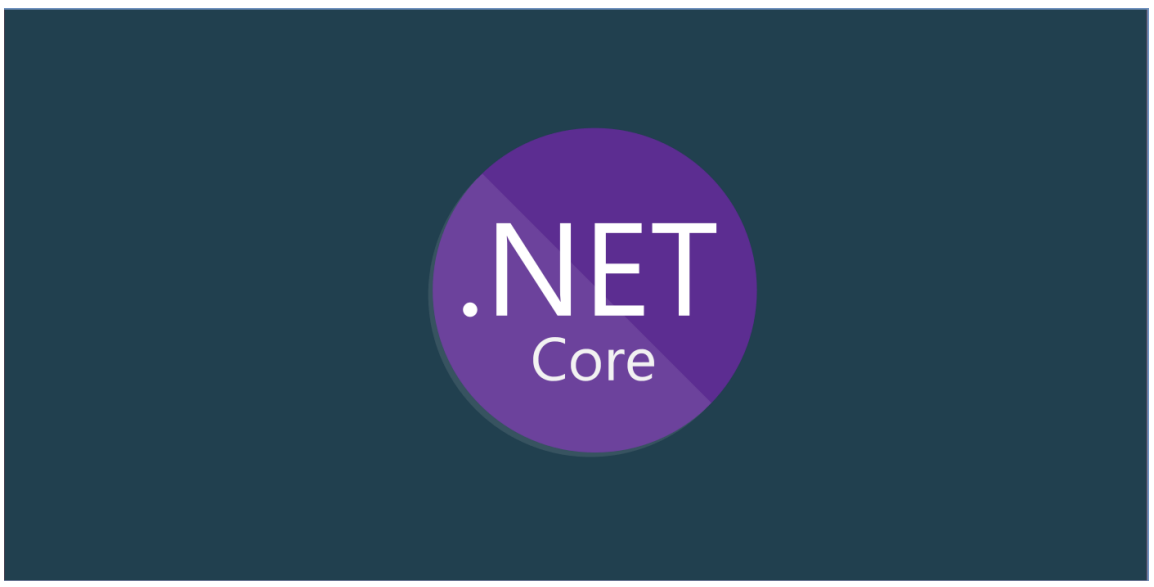

Implementing a Simple ETag for Response Caching Efficiency in an ASP.NET Core API

ASP.NET Core ETag • Posted 11 months ago

When it comes to developing applications with strict client and server components, where the client shall only focus on view logic while the data is supplied by the server via REST API endpoints, performance is the key. applications such as mobile, SPA or others which focus on view logic while relying on APIs for their data requirements, we would need the applications to load faster than their peers, along with mechanisms which can further complement load times such as lazy loading or caching less-frequently changing resources like assets or metadata files. And in such scenarios, in caching specifically, we would need the client to know when to refresh an already cached data and when to continue rendering it instead of polling on the API for changes.

Consider the scenario of a Table application which loads a huge grid of data from an API endpoint. And this data is cached for certain amount of time for faster loads and offline capabilities, and once the cache expires the client hits on the API endpoint for fresh dataset. Now what

SIMILAR ARTICLES

Exploring ASP.NET Core MVC - Understanding ViewBag and ViewData
ASP.NET Core

Exploring ASP.NET Core Fundamentals - Understanding ViewComponents
ASP.NET Core

Exploring ASP.NET Core Fundamentals - Understanding Singleton Transient and Scoped Service Lifetimes
ASP.NET Core

Exploring ASP.NET Core Fundamentals - Understanding Middlewares
ASP.NET Core

Exploring ASP.NET Core Fundamentals - Getting started with .NET Core CLI
ASP.NET Core

LABELS

ASP.NET CORE AWS FLUTTER DESIGN PATTERNS ANGULAR C# CONCEPTS DOCKER JWT ALGORITHMS IDENTITYSERVER4 JENKINS XUNIT SOLID PRINCIPLES AUTHENTICATION AZURE UNIT TESTING EXPRESS NODEJS UPGRADE TO .NET CORE 3.X AUTHORIZATION COGNITO CORE MYSQL BOOTSTRAP DYNAMODB ETAG GIT INTEGRATION TESTING

still fetch data from the API and then reload the cache with the same data which was already present in it. For this, we would need the client to tell the server in some way that it has some "version" of the data present with it, and the server should reply back if that "version" has become obsolete by the time of request or if the data is "not yet modified". This mechanism is provided by a request-response header communication between the client and the server by exchanging a specific string in between them; this is known as ETag.

Conceptualizing an ETag:

An ETag or an "*entity-tag*" is simply a string; which represents some "version" of the data that the client possesses, and the client conveys this "version" to the server to check whether any new "version" of it is already created at the server end. When there is no new "version" of the resource possessed by the client, the server replies back saying resource is **"304 Not Modified"**. If there is a new "version" available at the server end, the server replies back with the new resource to the client along **"200 OK"** response and a new ETag which represents this new "version" of the string for the client to convey from then onwards. This "entity-tag" is purely "opaque-tag", meaning there's nothing that can be concealed from the string about the resource.

The server sends the generated etag for a specific version of a resource in the response header "ETag". And the client stores and uses this header value for consecutive API requests by passing in this value under a request header "If-None-Match". When the server encounters this specific request header from the client, it matches this value to the value generated from the resource that is fetched currently at the server. If both match, it means that the data is unchanged and its not required for the client to update its localstore. The server conveys this by sending a response status code "304 Not Modified" with an empty response body. The client can then simply update its cache expiry for a little longer duration. This cycle continues untill the resource changes at the server end. This quite helps in saving the bandwith between the server and the client and helps client act faster, supplementing its user experience.

The Implementation Puzzle:

While the Web Specification guides only on what an ETag is supposed to do and how it must be exchanged between the client and the server, it doesn't specify anything regarding how an ETag must be generated at the server end; since its solely server-centric to be honest. Since the client has no clue about the string that it deemed to be an "ETag" and doesn't need to parse it for any information unlike a JWT token, the server can employ any kind of mechanism to implement this ETag. Most common and simple implementation would be to generate an ETag once the resource is generated at the API end and just before it is

sent out to the client, while there can be implementations basing on a real "version identifier" sort of implementations on the resource which is maintained at the backend data store.

Simple ETag Implementation by Response in an ASP.NET Core API:

Let's take the example of our ReadersAPI which returns a set of readers to the client. Now let's assume we would need this API endpoint to generate and send ETags to the calling client, to facilitate caching at the client end. For this, we can take help from an Action Filter which can apply on the resultant dataset "after execution".

Consider the API as below:

```
namespace ReadersApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ReaderController : ControllerBase
    {
        // invokes the ETag Action Filter
        // when the API is called
        [ETagFilter]
        [Route("all")]
        [HttpGet]
        public IEnumerable<Reader> GetReaders()
        {
            return ReaderStore.Readers.Where(x => x.Id != 0);
        }
    }
}
```

In order to attach an ETag to this response, let's define an ActionFilter that generates an ETag and attaches it to the response headers as below:

```
namespace ReadersApi.Providers
{
    // prevents the action filter methods to be invoked twice
    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
          AllowMultiple = false)]
    public class ETagFilter : ActionFilterAttribute
    {
        public override void OnActionExecuted(ActionExecutedContext context)
        {
            var request = context.HttpContext.Request;
            var response = context.HttpContext.Response;

            if (request.Method == HttpMethod.Get.Method &&
             response.StatusCode == (int)HttpStatusCode.OK)
            {
                var res = JsonConvert.SerializeObject(context.Result);

                // generate etag string
                // from the response body
                var etag = GenerateETag(res);

                //fetch etag from the incoming request header
                if (request.Headers.Keys.Contains(HeaderNames.IfNoneMatch))
                {
                    var incomingEtag =
                                request.Headers[HeaderNames.IfNoneMatch]
                                    .ToString();

                    // if both the etags are equal
                    // raise a 304 Not Modified Response
                    if (incomingEtag.Equals(etag))
                    {
                        context.Result =
                                new StatusCodeResult(
                                (int)HttpStatusCode.NotModified);
                    }
                }

                // add ETag response header
                response.Headers.Add(HeaderNames.ETag, new[] { etag });
            }

            base.OnActionExecuted(context);
        }

        private string GenerateETag(string response)
        {
                // mechanism to generate ETag string
              // for the response content
              // can be any mechanism chosen by the developer
        }
    }
}
```

Let's try to understand what is happening here. Since we have overridden the OnActionExecuted() method of the ActionFilterAttribute, the method is invoked only after the result is generated and before it is sent out to the user. Within this method, we generate ETag string based on the result content only for GET requests which are successful in execution (200 OK response). Here we just serialize the result and then generate ETag out of it (using any mechanism, be it hashing or encryption or any method of developer's choice). Then we check if there's already an etag being sent in the request. If there's one available we match it directly with the generated etag to check if the response is stale (unchanged). If they match, we

send out a Not Modified Status response directly. Else we add the generated etag to the response header and then leave the response to the client.

When we run this:

I. GET Request without ETag Request Header:

```
GET /api/reader/all
Host: localhost:5000
Content-Type: application/json
```

Response shall be:

```
[
{"id":1003,"userName":"reader1003","emailAddress":"reader1003@me.com"},
{"id":1002,"userName":"reader1002","emailAddress":"reader1002@me.com"}
]

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Server: Kestrel
Transfer-Encoding: chunked
ETag: eyJWYWx1ZSI6W3siSWQiOj...29kZSI6bnVsbH0=
```

II. GET Request with ETag in Request Header:

```
GET /api/reader/all
Host: localhost:5000
If-None-Match: eyJWYWx1ZSI6W3siSWQiOj...29kZSI6bnVsbH0=
Content-Type: application/json
```

Response shall be:

```
HTTP/1.1 304 Not Modified
Server: Kestrel
ETag: eyJWYWx1ZSI6W3siSWQiOj...29kZSI6bnVsbH0=
```

III. When the resource changes:

```
GET /api/reader/all
Host: localhost:5000
If-None-Match: eyJWYWx1ZSI6W3siSWQiOj...29kZSI6bnVsbH0=
Content-Type: application/json
```

Shall Return new Response and a new ETag:

```
[{"id":1003,"userName":"reader1003","emailAddress":"reader1003@me.com"}]

HTTP/1.1 200 OK
Date: Wed, 26 Feb 2020 17:14:46 GMT
Content-Type: application/json; charset=utf-8
Server: Kestrel
Transfer-Encoding: chunked
ETag: eyJWYWx1ZSI6IlVz...N0YXR1c0NvZGUiOm51bGx9
```

« Previous
Configuring Response Caching Headers in an ASP.NET Core API

Next »
C# Fundamentals - Using the yield keyword

This way we can implement a simple ETag mec
response content. Alternatively, we can impleme

Ram

I'm a full-stack developer and a software enthusiast who likes to play around with cloud and tech stack out of curiosity.