Scalable and Performant ASP.NET Core Web APIs: Client Caching

March 08, 2018

dotnet

This is yet another post in a series on creating performant and scalable web APIs using ASP.NET Core. In this post, we'll start to focus on caching. Often the slowest bit of a web API is fetching the data, so, if the data hasn't changed it makes sense to save it in a place that can be retrieved a lot quicker than a database or another API call.

We will focus on leveraging standard HTTP caching and in this post we'll focus on client side caching. There are 2 HTTP caching approaches: - **expiration** and **validation** ...

EXPIRATION

This is simplest approach and is where the cache expires after a certain amount of time. It also yields the best performance because the client does not need to make any request to the server if the resource is in the client cache. This is the best approach for data that rarely changes.

To implement this, we need to add the ASP.NET Core response caching middleware just before the MVC middleware is added:

```
public class Startup
{
```

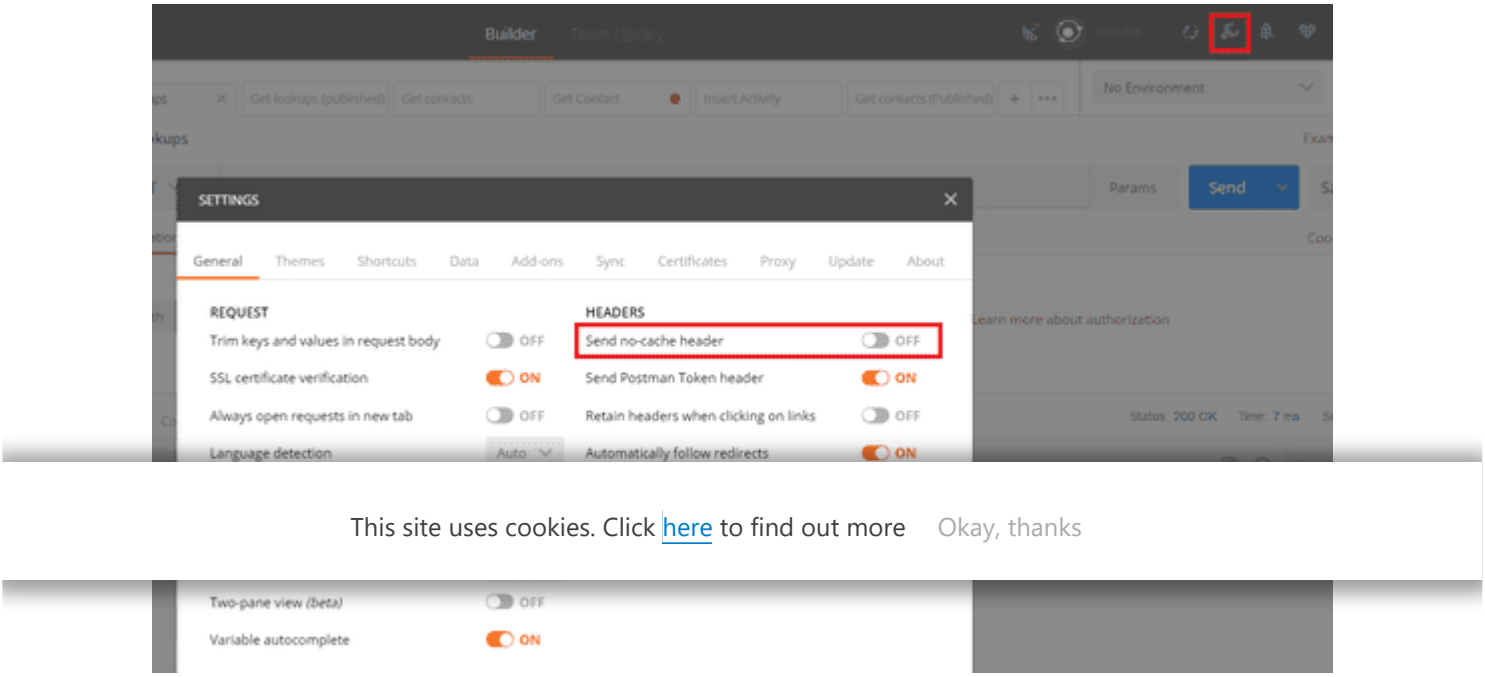
```
        ...
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddResponseCaching();
            services.AddMvc();
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            ...
            app.UseResponseCaching();
            app.UseMvc();
        }
    }
```



```
    [ResponseCache(Duration = 1800)]
    [HttpGet("lookups")]
    public IActionResult GetLookups()
    {
        var lookups = new Lookups();

        using (SqlConnection connection = new SqlConnection(_connectionString))
        {
            connection.Open();

            using (GridReader results = connection.QueryMultiple(@"
                    SELECT Type FROM AddressType
                    SELECT Type FROM EmailAddressType
                    SELECT Type FROM PhoneNumberType"))
            {
                lookups.AddressTypes = results.Read<string>().ToList();
                lookups.EmailAddressTypes = results.Read<string>().ToList();
                lookups.PhoneNumberTypes = results.Read<string>().ToList();
            }
        }

        return Ok(lookups);
    }
```

This sets the `Cache-Control` HTTP header:  Cache-Control HTTP header

Note: to test that Postman doesn't hit our code on the 2nd request (once the response has been cached), we need to make sure Postman doesn't send a "cache-control: no-cache" HTTP header:



If we put a breakpoint in our action method, after Postman has cached the response, we'll find the breakpoint is not hit if we invoke the request again.

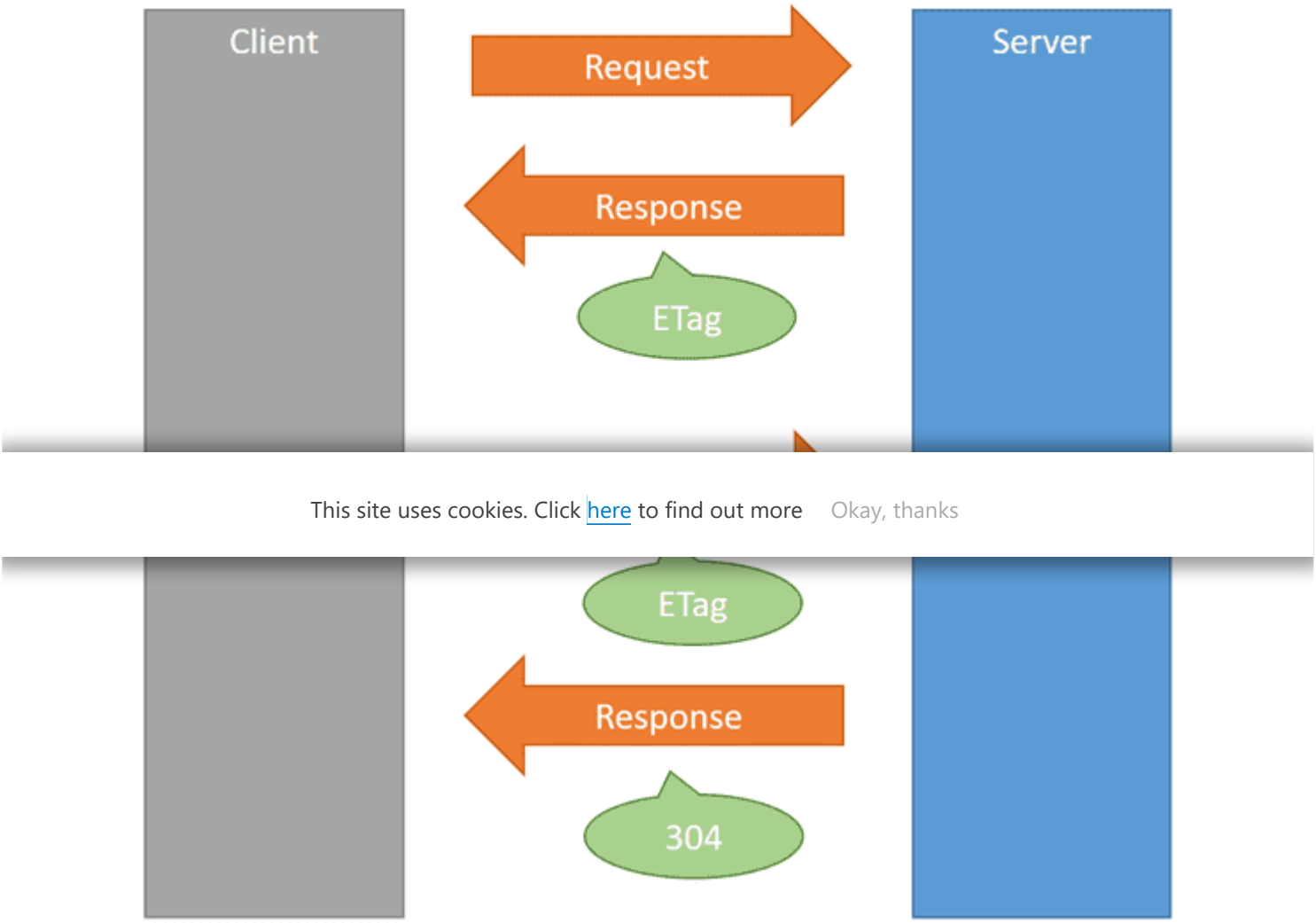
Neat!

VALIDATION

With the validation approach, the server returns a 304 (Not Modified) if the resource hasn't changed since the last request. This is the best approach for data that often changes. So, unlike the expiration approach, we still need to make a round trip, but the benefit of this approach is reduced network bandwidth.

A good mechanism of implementing validation based caching is to leverage ETags. With this approach, the server includes an ETag in the response for a resource which represents a unique value for the version of the resource. Clients then include the ETag value in subsequent requests to the resource (in a `If-None-Match` HTTP

header) that the client has in its cache. The server can then return a 304 if the ETag for the requested resource matches the supplied ETag.



So, let's implement an ETag for an action method for `GET api/contacts/{contactId}` continuing to use the Dapper as our data access library ...

We need to return the version of the contact record (`Contact.RowVersion`) that can be used as the ETag in our data access code. This leverages SQL Server's rowversion data type, so, this is a strong ETag.

In our action method, we grab the ETag from the `If-None-Match` HTTP header at the start of the method. At the end of the method, we return a 304 if the ETag in the `If-None-Match` HTTP header is the same as from the returned record.

```
[HttpGet("{contactId}")]
public IActionResult GetContact(string contactId)
{
    Contact contact = null;

    // Get the requested ETag
    string requestETag = "";
    if (Request.Headers.ContainsKey("If-None-Match"))
    {
        requestETag = Request.Headers["If-None-Match"].First();
    }

    // Get the contact from the database
    using (SqlConnection connection = new SqlConnection(_connectionString))
    {
        connection.Open();
```



```
                    FROM Contact
                    WHERE ContactId = @ContactId";

        contact = connection.QueryFirst<Contact>(sql, new { ContactId = contactId
    }

    // If no contact was found, then return a 404
    if (contact == null)
    {
        return NotFound();
    }

    // Construct the new ETag
    string responseETag = Convert.ToBase64String(contact.RowVersion);

    // Return a 304 if the ETag of the current record matches the ETag in the "If-
    if (Request.Headers.ContainsKey("If-None-Match") && responseETag == requestETa
    {
        return StatusCode((int)HttpStatusCode.NotModified);
    }

    // Add the current ETag to the HTTP header
    Response.Headers.Add("ETag", responseETag);

    return Ok(contact);
}
```

If we test this in Postman, without the `If-None-Match` HTTP header, we get the full record returned with the ETag.

On the second request, with the `If-None-Match` HTTP header we get a 304 with no response body.

The performance and scalability impact with what we have just done isn't greatly positive. In fact, the 2nd request (where we have the data cached on the client) was slower than the 1st request in the above timings. This is because we are still accessing the data from the database even though it is cached on the client. The only saving we are really making is the reduced size of the response, which isn't massive in our example. Maybe caching the data on the server will improve performance? Also, the amount of code we need to write each action method to facilitate ETags is a little worrying - we'll cover both these topics in the next post ...

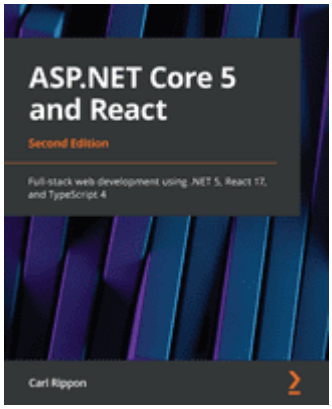
Did you find this post useful?

Let me know by sharing it on Twitter.

Click here to share this post on Twitter

If you to learn about using React with ASP.NET Core you might find my book useful:

ASP.NET Core 5 and React



Find out more

[Why React with ASP.NET Core?]

[What's new in ASP.NET Core 3 for React SPAs?]

[ASP.NET Core Logging with Serilog and SQL Server]

[Scalable and Performant ASP.NET Core Web APIs: Asynchronous Code]

[Integration Testing on ASP.NET Core Web API controllers with a SQL backend]

← Scalable and Performant ASP.NET Core Web APIs: Filtering and Searching

Scalable and Performant ASP.NET Core Web APIs: Server Caching →

Want more content like this?

Subscribe to receive notifications on new blog posts and courses

Email Required

First Name

Subscribe

Privacy Policy