25 CACHING

In this section, we are going to learn about caching resources. Caching can improve the quality and performance of our app a lot, but again, it is something first we need to look at as soon as some bug appears. To cover resource caching, we are going to work with HTTP Cache. Additionally, we are going to talk about cache expiration, validation, and cache-control headers.

25.1 About Caching

We want to use cache in our app because it can significantly improve performance. Otherwise, it would be useless. The main goal of caching is to eliminate the need to send requests towards the API in many cases and also to send full responses in other cases.

To reduce the number of sent requests, caching uses the **expiration mechanism**, which helps reduce network round trips. Furthermore, to eliminate the need to send full responses, the cache uses the **validation mechanism**, which reduces network bandwidth. We can now see why these two are so important when caching resources.

The cache is a separate component that accepts requests from the API's consumer. It also accepts the response from the API and stores that response if they are cacheable. Once the response is stored, if a consumer requests the same response again, the response from the cache should be served.

But the cache behaves differently depending on what cache type is used.

25.1.1 Cache Types

There are three types of caches: Client Cache, Gateway Cache, and Proxy Cache.

The client cache lives on the client (browser); thus, it is a private cache. It is private because it is related to a single client. So every client consuming our API has a private cache.

The gateway cache lives on the server and is a shared cache. This cache is shared because the resources it caches are shared over different clients.

The proxy cache is also a shared cache, but it doesn't live on the server nor the client side. It lives on the network.

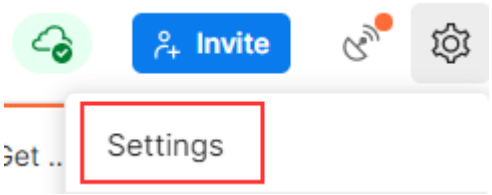
With the private cache, if five clients request the same response for the first time, every response will be served from the API and not from the cache. But if they request the same response again, that response should come from the cache (if it's not expired). This is not the case with the shared cache. The response from the first client is going to be cached, and then the other four clients will receive the cached response if they request it.

25.1.2 Response Cache Attribute

So, to cache some resources, we have to know whether or not it's cacheable. The response header helps us with that. The one that is used most often is Cache-Control: `Cache-Control: max-age=180`. This states that the response should be cached for 180 seconds. For that, we use the `ResponseCache` attribute. But of course, this is just a header. If we want to cache something, we need a cache-store. For our example, we are going to use Response caching middleware provided by ASP.NET Core.

25.2 Adding Cache Headers

Before we start, let's open Postman and modify the settings to support caching:



In the General tab under Headers, we are going to turn off the Send no-cache header:

Great. We can move on.

Let's assume we want to use the ResponseCache attribute to cache the result from the **GetCompany** action:

```
public class ResponseCacheAttribute : Attribute, IFilterFactory, IFilterMetadata, IOrderedFilter
{
    public int Duration
    public ResponseCacheLocation Location
    public bool NoStore
    public string? VaryByHeader
    public string[]? VaryByQueryKeys
    public string? CacheProfileName
    public int Order
    public bool IsReusable

    public CacheProfile GetCacheProfile(MvcOptions options)
    public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)

    public ResponseCacheAttribute()
}
```

It is obvious that we can work with different properties in the ResponseCache attribute — but for now, we are going to use **Duration** only:

```
[HttpGet("{id}", Name = "CompanyById")]
[ResponseCache(Duration = 60)]
public async Task<IActionResult> GetCompany(Guid id)
```

And that is it. We can inspect our result now:

You can see that the Cache-Control header was created with a public cache and a duration of 60 seconds. But as we said, this is just a header; we need a cache-store to cache the response. So, let's add one.

25.3 Adding Cache-Store

The first thing we are going to do is add an extension method in the `ServiceExtensions` class:

```
public static void ConfigureResponseCaching(this IServiceCollection services) =>
services.AddResponseCaching();
```

We register response caching in the IOC container, and now we have to call this method in the `Program` class:

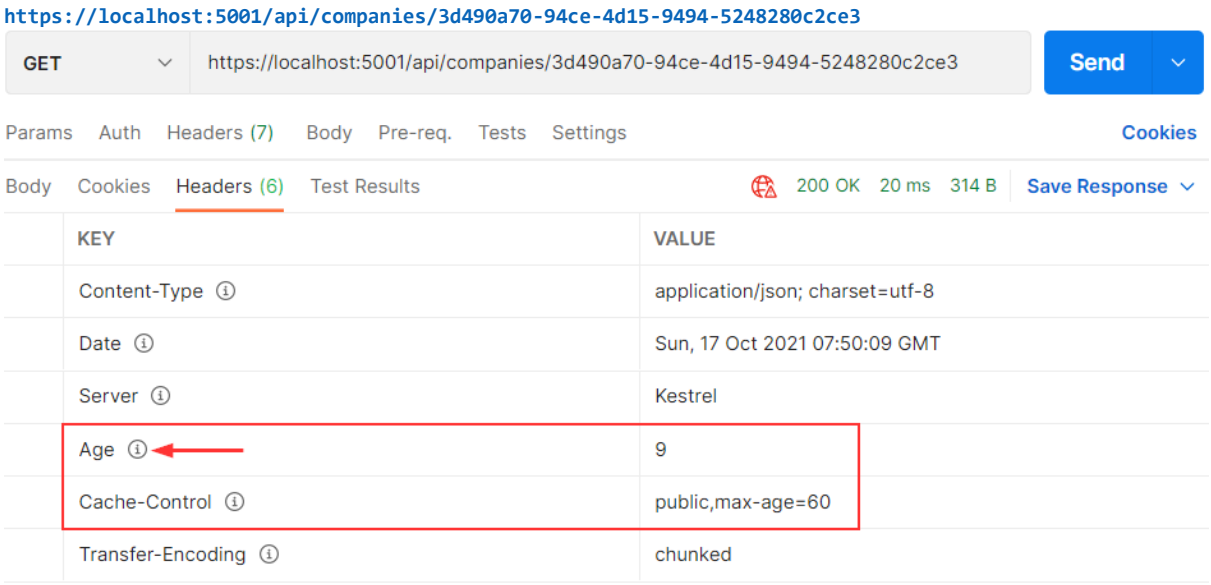
```
builder.Services.ConfigureResponseCaching();
```

Additionally, we have to add caching to the application middleware right below `UseCors()` because Microsoft recommends having `UseCors` before `UseResponseCaching`, and as we learned in the section 1.8, order is very important for the middleware execution:

```
app.UseCors("CorsPolicy");
app.UseResponseCaching();
```

Now, we can start our application and send the same `GetCompany` request. It will generate the Cache-Control header. After that, before 60

seconds pass, we are going to send the same request and inspect the headers:



You can see the additional `Age` header that indicates the number of seconds the object has been stored in the cache. Basically, it means that we received our second response from the cache-store.

Another way to confirm that is to wait 60 seconds to pass. After that, you can send the request and inspect the console. You will see the SQL query generated. But if you send a second request, you will find no new logs for the SQL query. That's because we are receiving our response from the cache.

Additionally, with every subsequent request within 60 seconds, the Age property will increment. After the expiration period passes, the response will be sent from the API, cached again, and the Age header will not be generated. You will also see new logs in the console.

Furthermore, we can use cache profiles to apply the same rules to different resources. If you look at the picture that shows all the properties we can use with `ResponseCacheAttribute`, you can see that there are a lot of properties. Configuring all of them on top of the action or controller

could lead to less readable code. Therefore, we can use **CacheProfiles** to extract that configuration.

To do that, we are going to modify the **AddControllers** method:

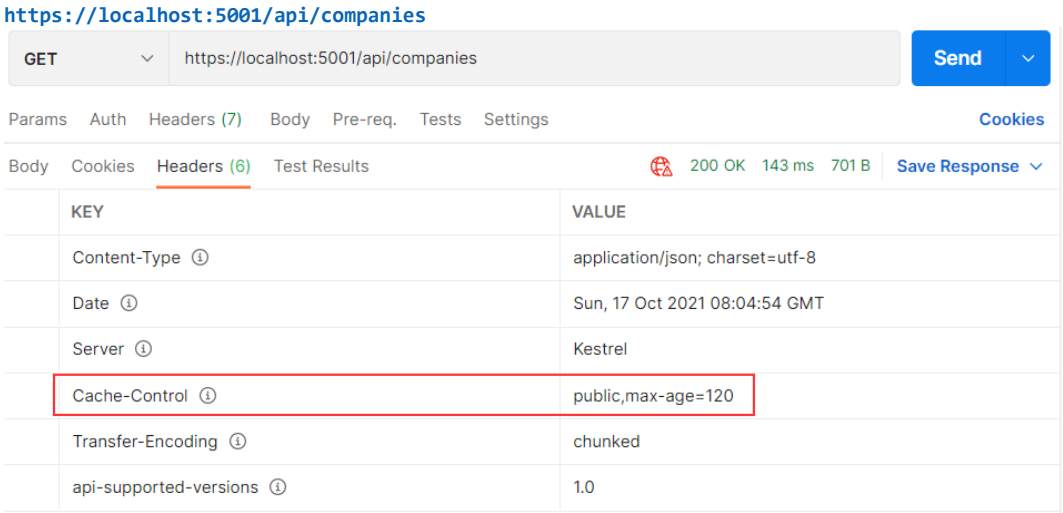
```
builder.Services.AddControllers(config =>
{
       config.RespectBrowserAcceptHeader = true;
       config.ReturnHttpNotAcceptable = true;
       config.InputFormatters.Insert(0, GetJsonPatchInputFormatter());
       config.CacheProfiles.Add("120SecondsDuration", new CacheProfile { Duration =
120 });
}).
```

We only set up Duration, but you can add additional properties as well. Now, let's implement this profile on top of the Companies controller:

```
[Route("api/companies")]
[ApiController]
[ResponseCache(CacheProfileName = "120SecondsDuration")]
```

We have to mention that this cache rule will apply to all the actions inside the controller except the ones that already have the ResponseCache attribute applied.

That said, once we send the request to **GetCompany**, we will still have the maximum age of 60. But once we send the request to **GetCompanies**:



There you go. Now, let's talk about new Output caching introduced in .NET 7.

25.4 Output Caching

Output caching is a mechanism for storing the output of a client's request and serving the stored result on future requests. This significantly improves an application's responsiveness by providing a faster response to client requests and saving on repeated processing of the same output. Caching is not a new concept in ASP.NET Core as we already have response caching features. But this newly designed Output Caching API (introduced in .NET 7) opens up a new horizon of possibilities with caching.

25.4.1 Differences Between Output and Response Caching

So why the need for another caching mechanism when one already exists? The answer lies in the limitations of Response Caching and the ever-increasing demand for a newer caching API to tackle newer challenges. Response Caching works through a set of standard HTTP cache headers where both **client and server play their roles**. The server application emits responses with appropriate headers according to cache configurations, and the **client conforms to those headers** to fetch the cached response. The client however can bypass the caching by using a no-cache header.

Output Caching is a different beast. Instead of using cache headers from client requests, the caching decision is **solely made by the server application.** The client is not supposed to know whether it's receiving a cached response. On one side, this new API means leaning toward a more practical approach. On the other side, this comes with some advanced capabilities. For example, we can configure resource locking - a true solution to prevent cache stampedes and thundering herds. Other features include:

- Support for custom caching stores like Redis
- Caching of cookies and headers

- Caching authenticated content
- Tagging of cache contents and invalidating as a group
- Purging cache
- Delayed caching
- Partial caching or donut caching

By default, output caching follows these rules:

- Only HTTP 200 responses are cached.
- Only HTTP GET or HEAD requests are cached.
- Responses that set cookies aren't cached.
- Responses to authenticated requests aren't cached.

25.5 Using Output Caching In Our App

Before we start with the examples, let's comment out all the response caching attributes in the `CompaniesController` and the line in the `Program` class where we configure `CacheProfiles` for the response caching.

After we do that, we are ready to use the output caching examples.

To register the output caching with stores in our app, we have to modify the `ConfigureResponseCaching` method in the `ServiceExtensions` class:

```
public static void ConfigureOutputCaching(this IServiceCollection services) =>
services.AddOutputCache();
```

Here, we modify the name of the method and then use the `AddOutputCache` method to register the output caching mechanism.

Next, we have to modify the call to this configure method in the Program class:

```
builder.Services.ConfigureOutputCaching();
```

And also call the `UseOutputCache` bellow the `UseCors` method to add the middleware for caching:

```
//app.UseResponseCaching();
app.UseOutputCache();
```

That said, let's do the same thing as we did with the `ResponseCache` attribute using the `GetCompany` action:

```
[HttpGet("{id:guid}", Name = "CompanyById")]
//[ResponseCache(Duration = 60)]
[OutputCache(Duration = 60)]
public async Task<IActionResult> GetCompany(Guid id)
{
   ...
}
```

The `OutputCache` attribute contains the properties that `ResponseCache` has, so we can use the `Duration` property this time as well.

Now, if we send the same GetCompany request from Postman, we will find a bit different headers:

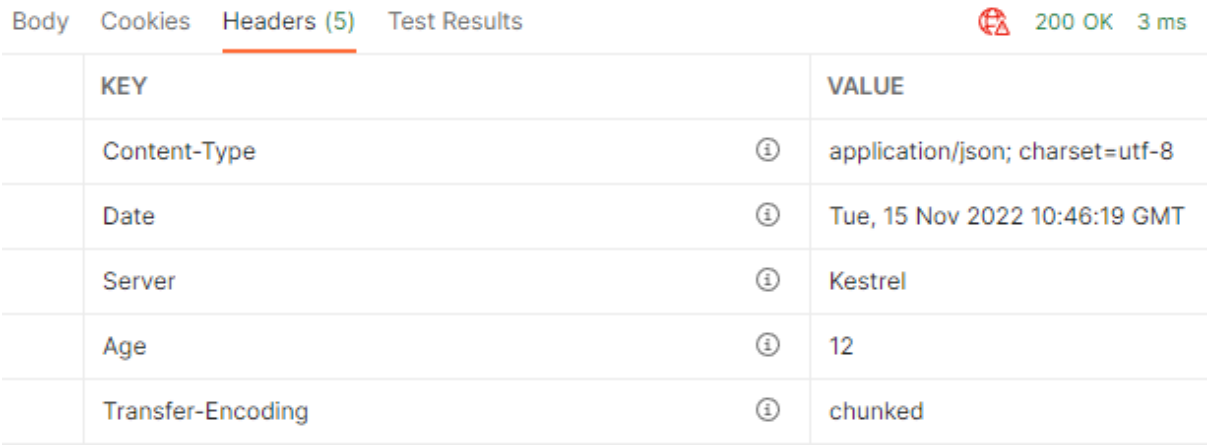
https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce3

Body Cookies Headers (5) Test Results 200 OK 59 ms 301 B

KEY	VALUE
Content-Type	application/json; charset=utf-8
Date	Tue, 15 Nov 2022 10:42:17 GMT
Server	Kestrel
Transfer-Encoding	chunked
api-supported-versions	1.0

We don’t have the `Cache-Control` header here anymore. That’s because the server doesn’t have to inform the client about the max-age directive of the response.

But, if we send the same request before 60 seconds pass, we will see the `Age` header for sure:



25.5.1 Using Policies With Output Caching

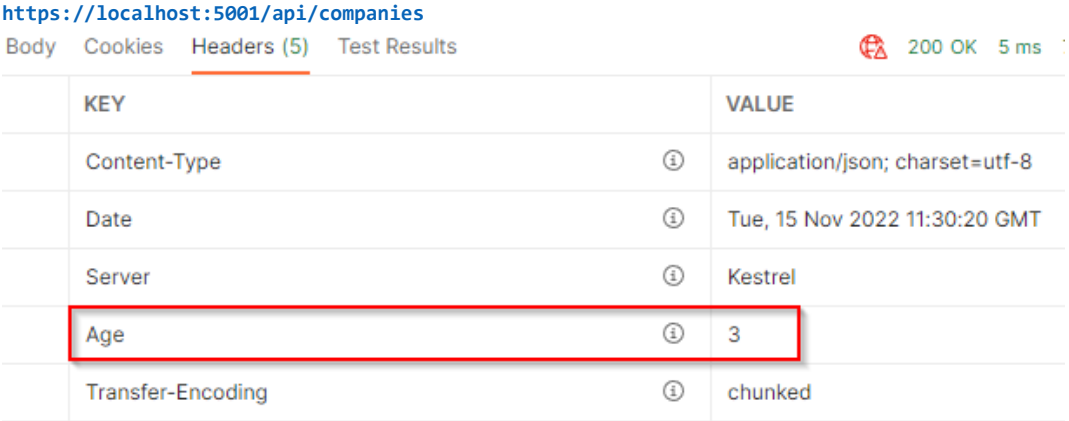
In a response caching section, we've used the profiles to set a profile and use it with the ResponseCache attribute. We can do the same with output caching, just this time, we have to use policies.

To use the policies, we have to modify the `AddOutputCache` method:

```
public static void ConfigureOutputCaching(this IServiceCollection services) =>
        services.AddOutputCache(opt =>
        {
            opt.AddBasePolicy(bp => bp.Expire(TimeSpan.FromSeconds(10)));
        });
```

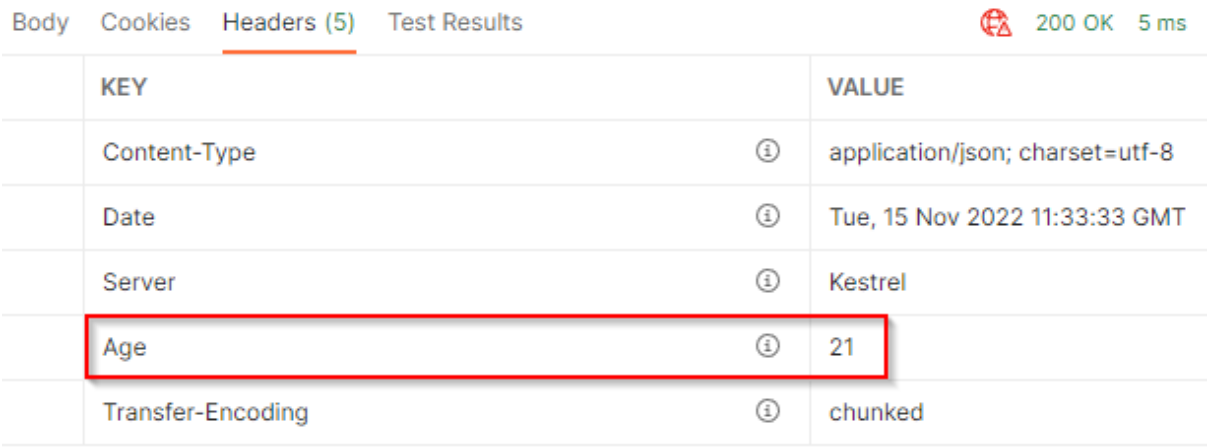
By using the `AddBasePolicy` method, we apply this base policy to all the endpoints in our controllers. We can confirm that.

Currently, we are not using any caching attribute with the `GetCompanies` action. But, if we send a request to that endpoint, and then another before 10 seconds pass, we will see the Age header:



Of course, after 10 seconds, our app will fetch a new result from the database.

On the other hand, if we send requests to the `GetCompany` endpoint, which already has the `OutputCache` attribute applied, we will see that the base policy doesn't affect it:



The value of the `Age` header is higher than 10, which means that using attributes override the base policy we just configured.

Besides the base policies, we can configure the named policies, which we have to apply to specific endpoints.

To do that, we can use the `AddPolicy` method:

```
public static void ConfigureOutputCaching(this IServiceCollection services) =>
    services.AddOutputCache(opt =>
    {
        //opt.AddBasePolicy(bp => bp.Expire(TimeSpan.FromSeconds(10)));
        opt.AddPolicy("120SecondsDuration", p => p.Expire(TimeSpan.FromSeconds(120)));
    });
```

Now, we can apply it to the controller or any specific action:

```
[Route("api/companies")]
[ApiController]
//[ResponseCache(CacheProfileName = "120SecondsDuration")]
[OutputCache(PolicyName = "120SecondsDuration")]
public class CompaniesController : ControllerBase
```

Even though we applied this policy to the entire controller, the `GetCompany` action will be cached for 60 seconds while other GET actions will be cached for 120 seconds.

25.5.2 Output Cache Keys

To show different keys that we can use with output caching, we are not going to modify our existing app but rather show some dummy examples so you could see how those keys are applied. They are pretty easy to use.

If, for example, we have created a base caching policy but don't want to use caching mechanism on certain actions, we can use the `NoStore` property of the `OutputCache` attribute:

```
[HttpGet("output-nocache")]
[OutputCache(NoStore = true)]
public IActionResult NonCachedOutput()
{
    return Ok($"Output was generated at {DateTime.Now}");
}
```

We can also use different “Vary” keys with output caching.

For example, we can enable cache mechanism for the query string parameter by using the `VaryByQueryKeys` property:

```
[HttpGet("output-varybykey")]
[OutputCache(VaryByQueryKeys = new[] { nameof(firstKey) })]
public IActionResult VaryByKey(string firstKey, string secondKey)
{
    return Ok($"{firstKey} {secondKey} - retrieved at {DateTime.Now}");
}
```

Here, we cache our response only based on the `firstKey` parameter. This means that we can change the second key as much as we want, and the response will be cached (for 60 seconds as this is a default period for caching - we didn't provide the `Duration` property). But, as soon as we change the `firstKey` value, our response will be generated again and then cached.

Of course, if we want to add `Duration` as well, we can do that inside the `OutputCache` attribute:

```
[HttpGet("output-varybykey")]
[OutputCache(VaryByQueryKeys = new[] { nameof(firstKey) }, Duration = 10)]
public IActionResult VaryByKey(string firstKey, string secondKey)
```

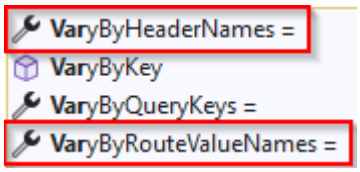
But, a much better way would be to create a policy with multiple rules:

```
services.AddOutputCache(opt =>
{
    opt.AddPolicy("QueryParamDuration", p =>
        p.Expire(TimeSpan.FromSeconds(10))
        .SetVaryByQuery("firstKey"));
});
```

And then use this policy:

```
[HttpGet("output-varybykey")]
[OutputCache(PolicyName = "QueryParamDuration")]
public IActionResult VaryByKey(string firstKey, string secondKey)
```

We should also be aware that next to this "vary" key, we have some additional ones that we can use similarly to the previous one:



25.5.3 Caching Revalidation

Cache revalidation means the server can return a `304 Not Modified` HTTP status code instead of the full response body. With that status code our server informs the client that the response to the request is unchanged from what the client previously received.

To see this in action, we can slightly modify the `GetCompany` action:

```
[HttpGet("{id:guid}", Name = "CompanyById")]
//[ResponseCache(Duration = 60)]
[OutputCache(Duration = 60)]
public async Task<IActionResult> GetCompany(Guid id)
{
    var company = await _service.CompanyService.GetCompanyAsync(id, trackChanges:
false);

    var etag = $"\"{Guid.NewGuid():n}\"";
    HttpContext.Response.Headers.ETag = etag;

    return Ok(company);
}
```

As soon as request arrives, the server creates an `etag` value and adds it to the `ETag` header.

Now, if we send a request to this endpoint, we will see a new header value:

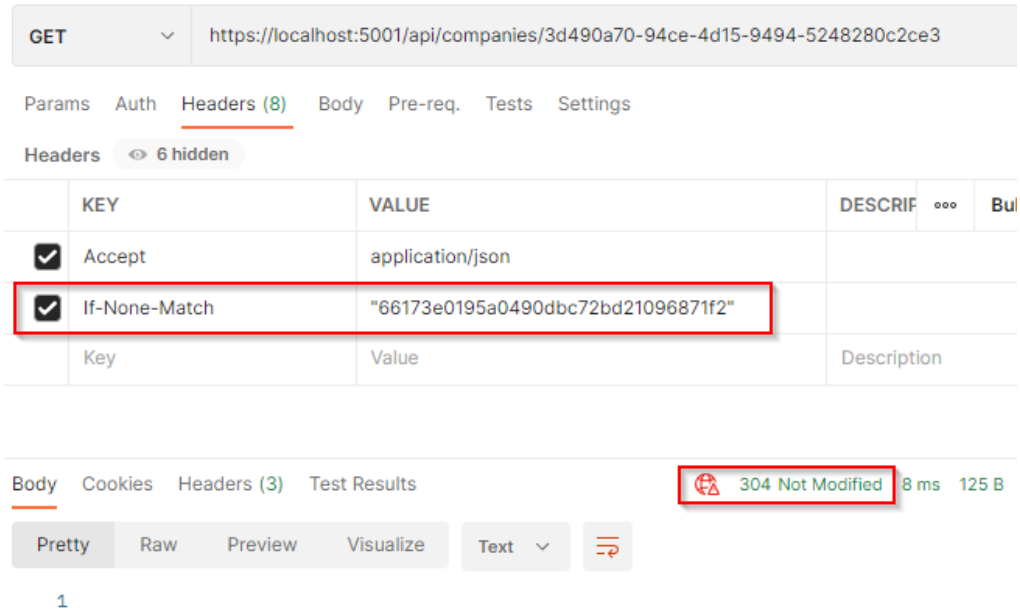
`https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce3`

KEY	VALUE
Content-Type	application/json; charset=utf-8
Date	Tue, 15 Nov 2022 13:22:55 GMT
Server	Kestrel
ETag	"66173e0195a0490dbc72bd21096871f2"
Transfer-Encoding	chunked
api-supported-versions	1.0

If we send another one before 60 seconds expire, we will find the same ETag value and the Age header as well. Also, pay attention that the status code of the response is 200 OK.

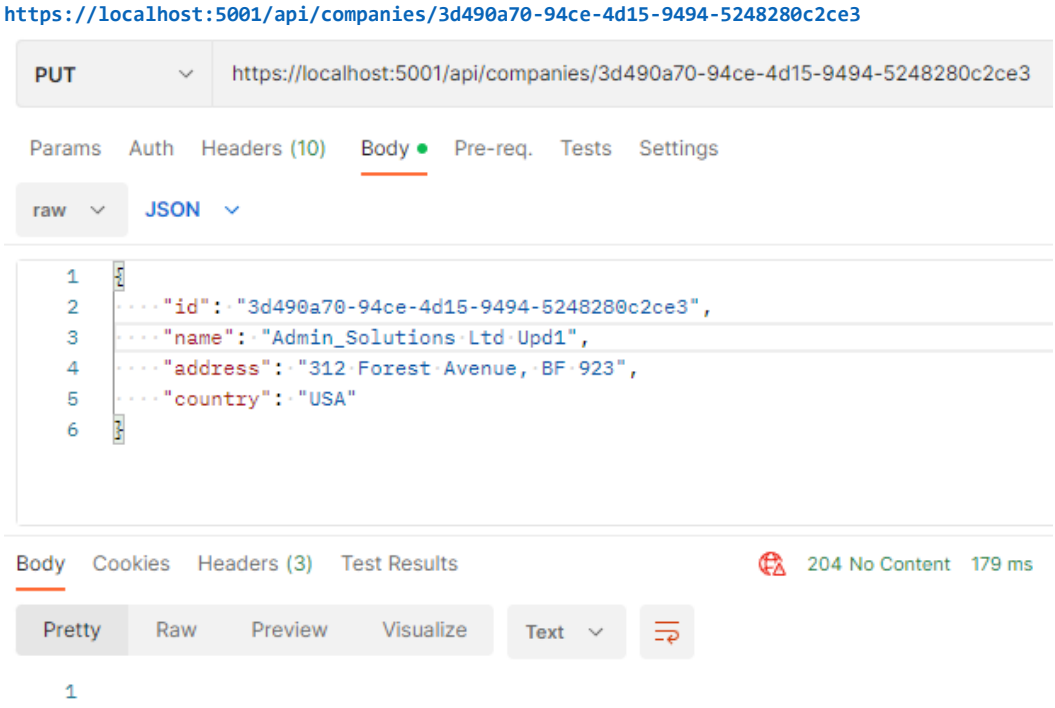
Now, as soon as we send another request to the same endpoint, but this time with the `If-None-Match` header with the same value as Etag's, we will see a different status code:

`https://localhost:5001/api/companies/3d490a70-94ce-4d15-9494-5248280c2ce3`

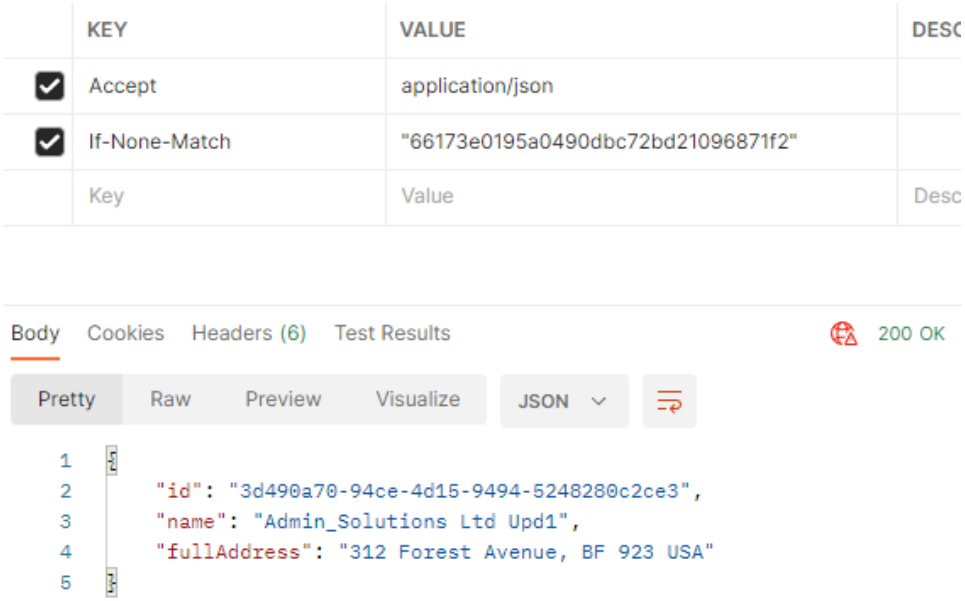


We get this status code because the resource we are fetching is the same.

Now, if we send a PUT request to update that company:



And then, while caching session is still active (no more than 60 seconds passed) we send another GET request with the same `If-None-Match` value:



We can see that we get 200 OK response and not 304 because our company is modified now.

Also, the ETag's value in the response is different:

KEY	VALUE
Content-Type	application/json; charset=utf-8
Date	Tue, 15 Nov 2022 13:24:17 GMT
Server	Kestrel
ETag	"a4b77f12ceae4fe181a531b3144961f4"
Transfer-Encoding	chunked
api-supported-versions	1.0

There are a lot more functionalities that output caching provides for us, but those we covered here should give you a good understanding of how output caching works and how to use it.