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.

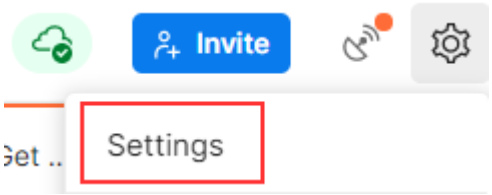
With the private cache, if five clients request the same response for the first time, every response will be served from the API and not from the cache. But if they request the same response again, that response should come from the cache (if it's not expired). This is not the case with the shared cache.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

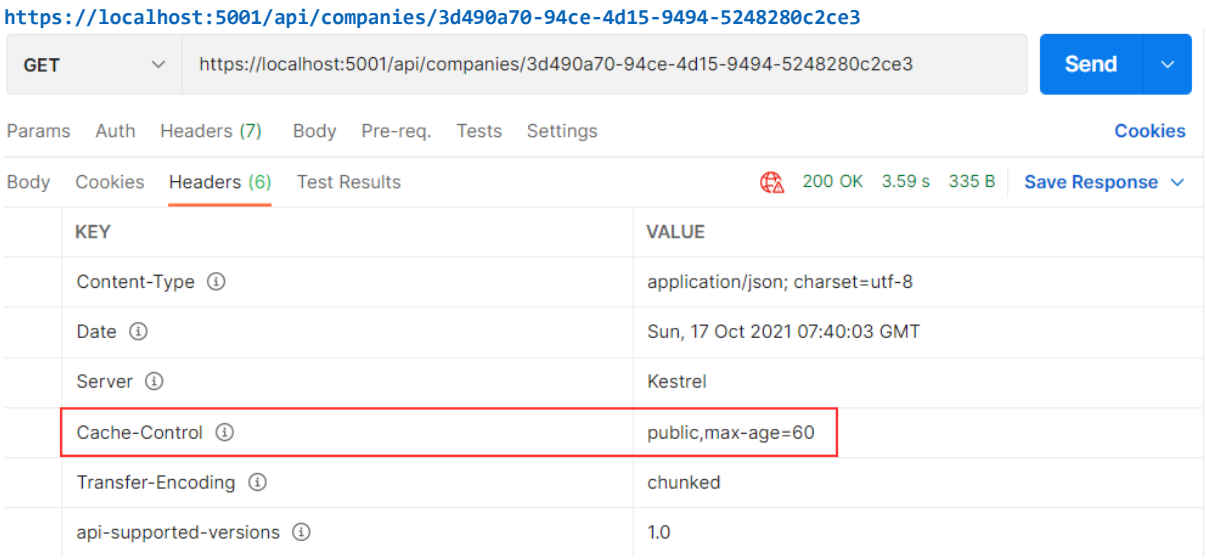
Let's assume we want to use the ResponseCache attribute to cache the result from the `GetCompany` action:

```
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:

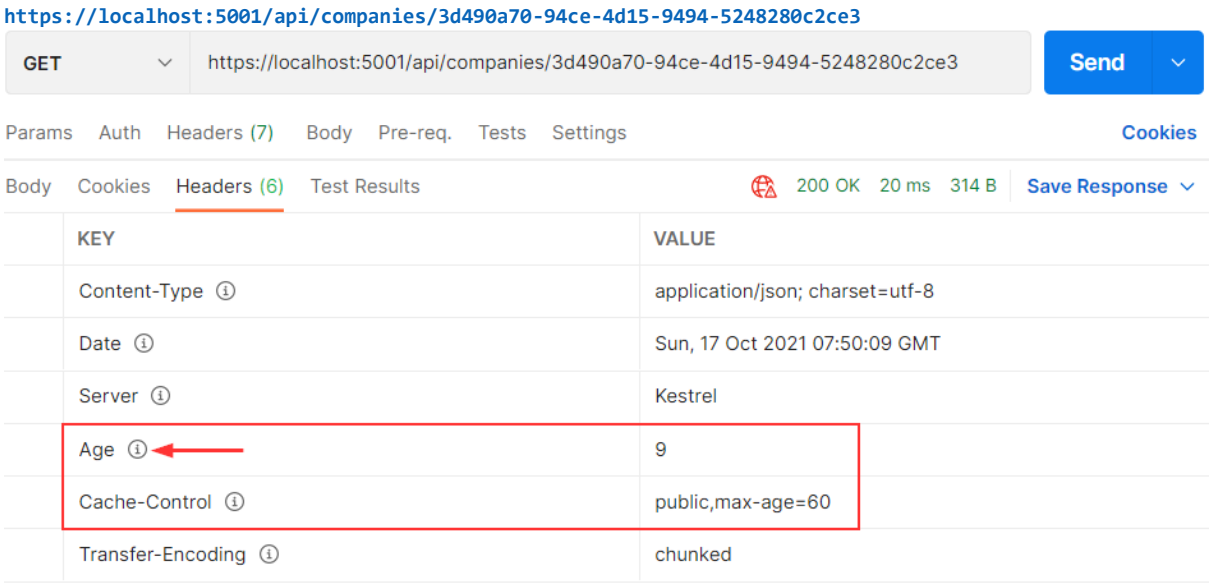
```
builder.Services.ConfigureResponseCaching();
```

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:

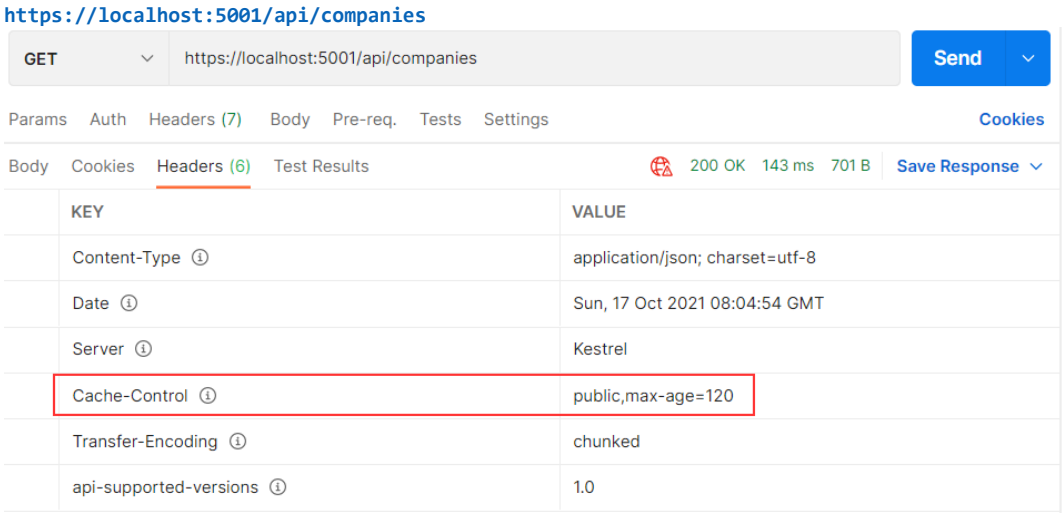
```
builder.Services.AddControllers(config =>
{
       config.RespectBrowserAcceptHeader = true;
       config.ReturnHttpNotAcceptable = true;
       config.InputFormatters.Insert(0, GetJsonPatchInputFormatter());
       config.CacheProfiles.Add("120SecondsDuration", new CacheProfile { Duration =
120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

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**:

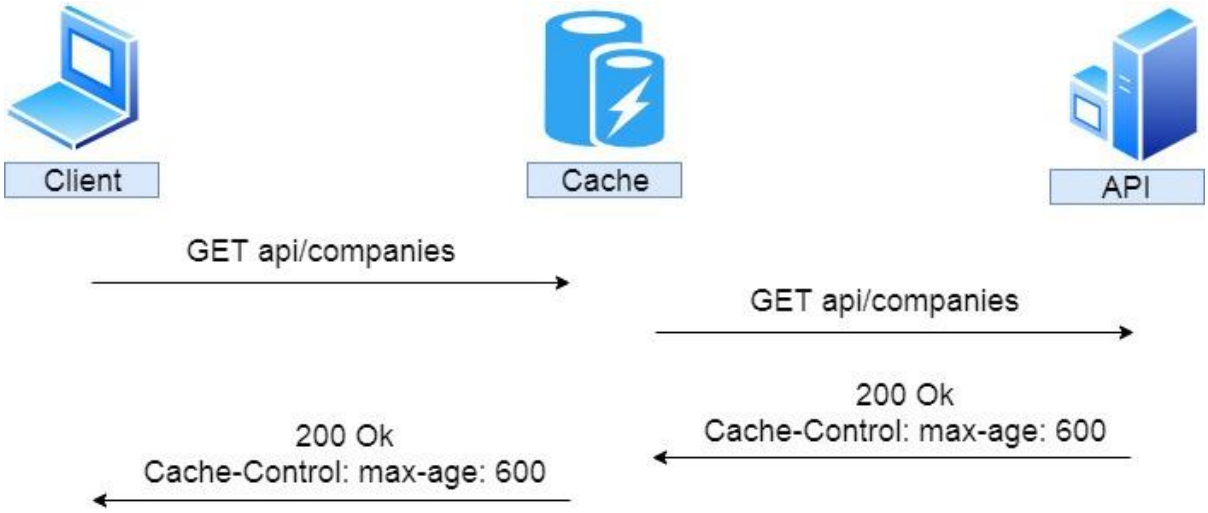


There you go. Now, let's talk some more about the Expiration and Validation models.

25.4 Expiration Model

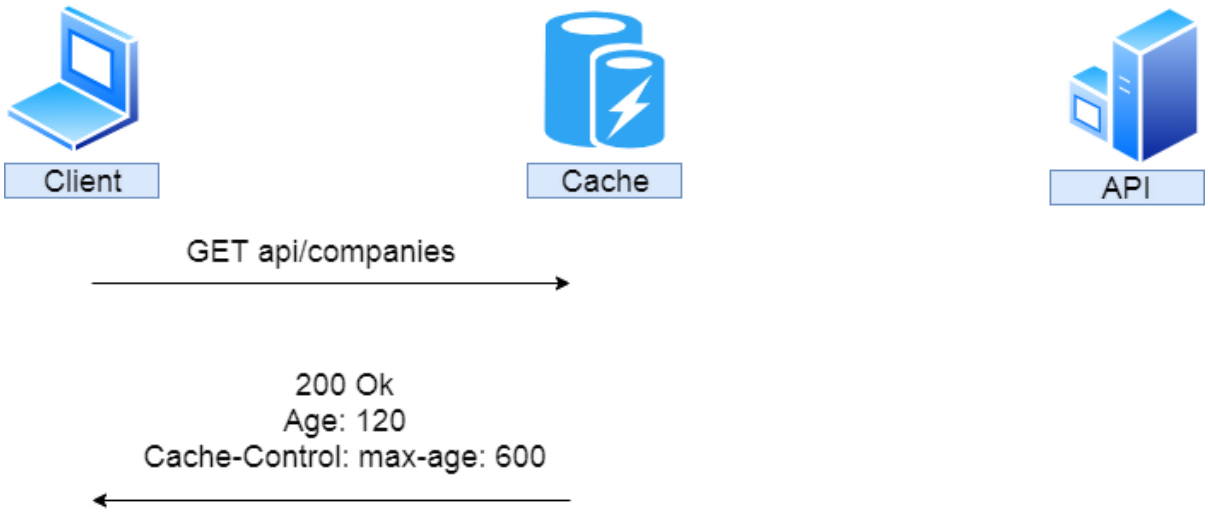
The expiration model allows the server to recognize whether or not the response has expired. As long as the response is fresh, it will be served from the cache. To achieve that, the Cache-Control header is used. We have seen this in the previous example.

Let's look at the diagram to see how caching works:

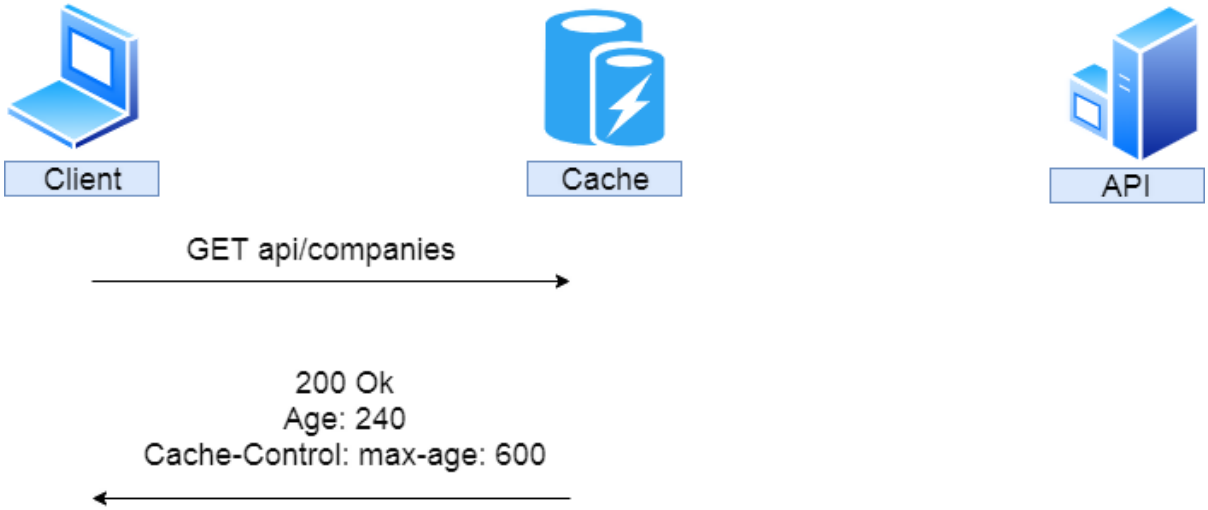


So, the client sends a request to get companies. There is no cached version of that response; therefore, the request is forwarded to the API. The API returns the response with the Cache-Control header with a 10-minute expiration period; it is being stored in the cache and forwarded to the client.

If after two minutes, the same response has been requested:



We can see that the cached response was served with an additional Age header with a value of 120 seconds or two minutes. If this is a private cache, that is where it stops. That's because the private cache is stored in the browser and another client will hit the API for the same response. But if this is a shared cache and another client requests the same response after an additional two minutes:



The response is served from the cache with an additional two minutes added to the Age header.

We saw how the Expiration model works, now let's inspect the Validation model.