Tutorial for Class number 9

This exercise intends to develop a web application to test the use of cache settings in HTTP responses**.**

In a second step, the exercise intends to define and apply customized action filters and examples of how these filters work.

First step – Creating a project without authentication

- Create a new "ASP.NET Core Web App (Model-View-Controller)" project.

  Create a **TestCacheController** controller that will be used to test different cache configurations.

  Create the **OnClient** action with a **20**-second **private** cache setting.

```
[ResponseCache(Duration =20,Location =ResponseCacheLocation.Client)]
0 references
public IActionResult OnClient()
{
    return View();
}
```

  With the following view

```
@{
    ViewData["Title"] = "OnClient";
}

<h1>OnClient</h1>

This view was built at @DateTime.Now.ToLongTimeString()
```

  Create the **OnDownStream** action with a **40**-second **public** cache setting.

```
[ResponseCache(Duration = 40, Location = ResponseCacheLocation.Any)]
0 references
public IActionResult OnDownStream()
{
    return View();
}
```

With the following view.

```
@{
    ViewData["Title"] = "OnDownStream";
}

<h1>OnDownStream</h1>

This view was built at @DateTime.Now.ToLongTimeString()
```

Although it is not necessary to use these cache configurations, we can configure the system to use cache in Middleware.
For more information see:
https://learn.microsoft.com/en-us/aspnet/core/performance/caching/middleware?view=aspnetcore-8.0

Configure the use of the cache in the **Program.cs** file.

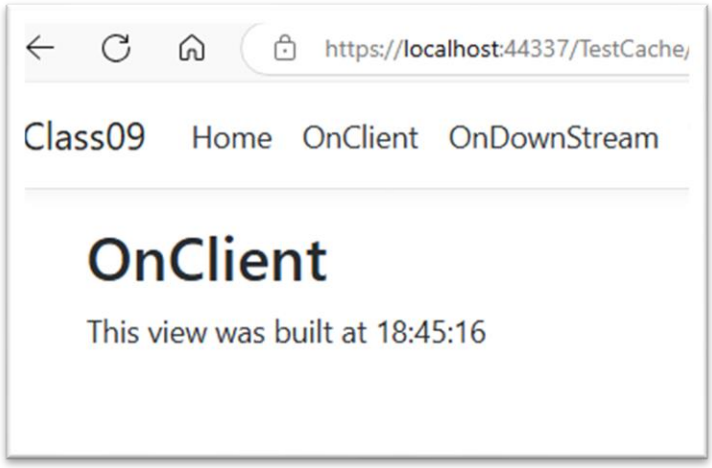
```
builder.Services.AddControllersWithViews();

builder.Services.AddResponseCaching(option =>
{
    // some default values can be changed
    option.SizeLimit = 200; // default is 100 MB
    option.MaximumBodySize = 20; // default value is 64 MB
});
```

```
app.UseStaticFiles();

app.UseResponseCaching();

app.UseRouting();
```

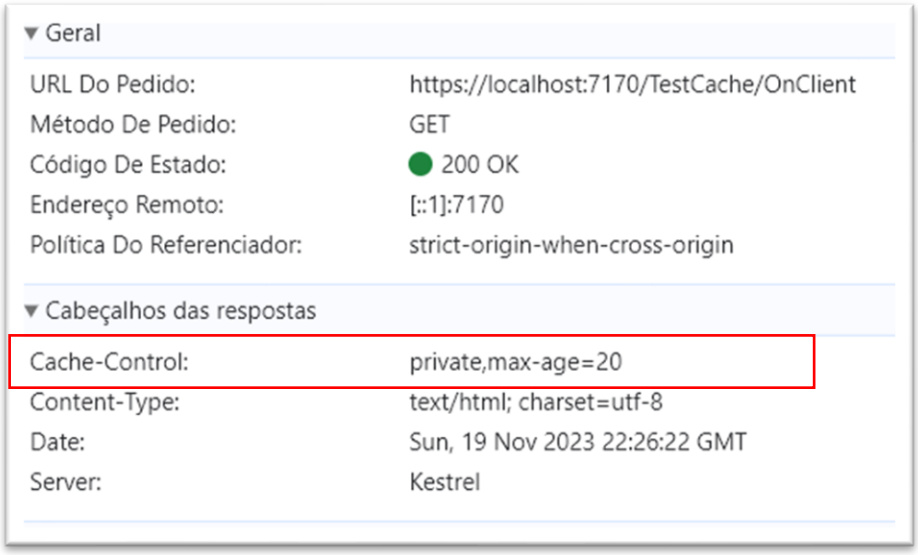
To facilitate access to the implemented resources, add two options to the menu in the file **_Layout.cshtml**.

```
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
</li>
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-controller="TestCache" asp-action="OnClient">OnClient</a>
</li>
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-controller="TestCache" asp-action="OnDownStream">OnDownStream</a>
</li>
```

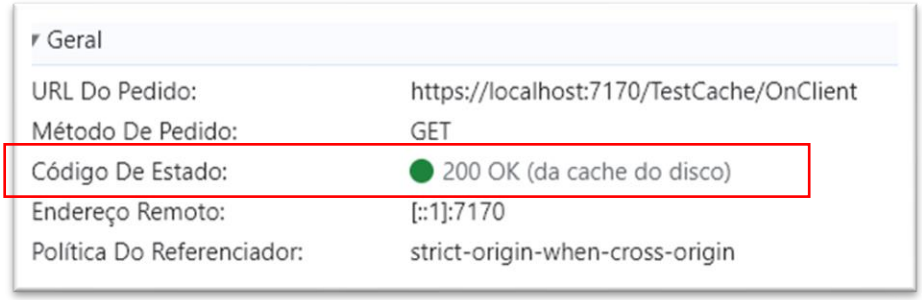
Try both features of the application by viewing the results (as shown in the following example):



In the *headers* resulting from the first order we can see:



In the following requests, the result indicates the use of the cached content (proven in all requests made until the cached content expires - 20 seconds).



After 20 seconds a new content is provided (confirm with the time shown in the view and with the response headers).

Note that the **If-Not-Modified-Since** header is not used by the browser (when content expires) because in the initial response the server does not sent the **Last-Modified** header.
Despite in the dynamically generated content this header information may not make sense, in the static content it can be important. Later on we will see how to include this information in HTTP messages.

Use predefined cache profiles.

Change the **Program.cs** file to create cache profiles.
The settings are created as options in the project's MVC structure.
Change the call to the **AddControllersWithViews** method to contain the following code:

```
builder.Services.AddControllersWithViews(options =>
{
    options.CacheProfiles.Add("TestCacheProfile",
        new Microsoft.AspNetCore.Mvc.CacheProfile()
        {
            Duration = 20,
            Location = Microsoft.AspNetCore.Mvc.ResponseCacheLocation.Any,
            VaryByHeader = "User-Agent"
        });
});
```

This profile defines a cache in the browser with a duration of 20 seconds and depending on the value of the “User-Agent” header.

Add to the **TestCacheController** controller an action named **WithHeaders**, configured with the cache profile previously created.

```
[ResponseCache(CacheProfileName = "TestCacheProfile")]
0 references
public IActionResult WithHeaders()
{
    return View();
}
```

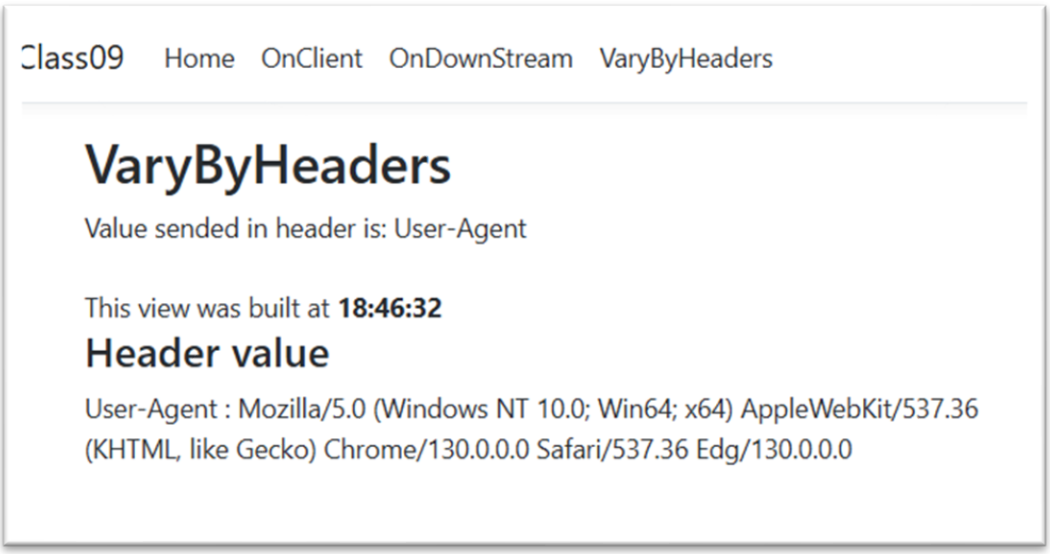
And the respective view

```
@{
    ViewData["Title"] = "WithHeaders";
}

<h1>VaryByHeaders</h1>

Value sended in header is: @Context.Response.Headers["Vary"]
<br /><br />
This view was built at <strong>@DateTime.Now.ToLongTimeString()</strong>
<br />

<h3>Header value</h3>
@Context.Response.Headers["Vary"] : @Context.Request.Headers[Context.Response.Headers["Vary"]]
```

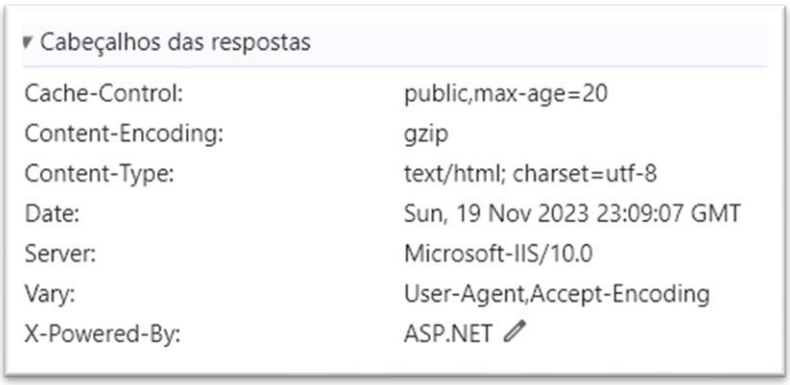
Add the navigation option to the **_Layout.cshtml** file menu and test the feature.

```
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-controller="TestCache" asp-action="WithHeaders">VaryByHeaders</a>
</li>
```

Try the feature simultaneously using two different browsers (for example Chrome and Firefox).



In addition to the headers seen in the previous step, we have the inclusion of the **Vary** header value in the responses.

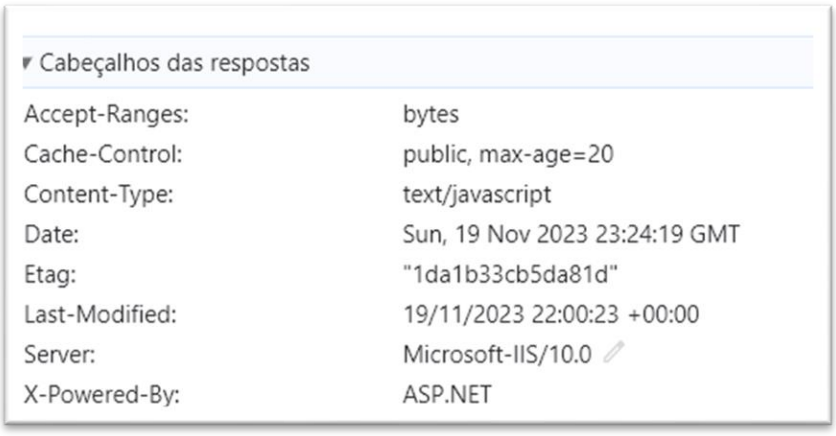


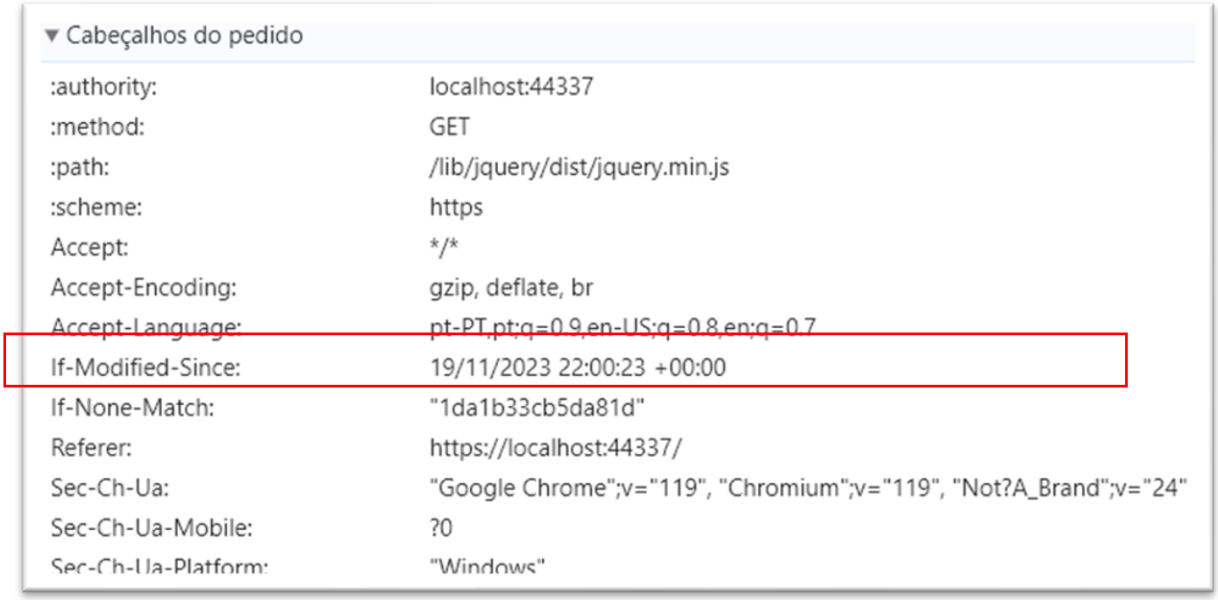
Add a configuration for the static content cache to the project. In this configuration we will add the **Last-Modified** header with the value from the requested file.

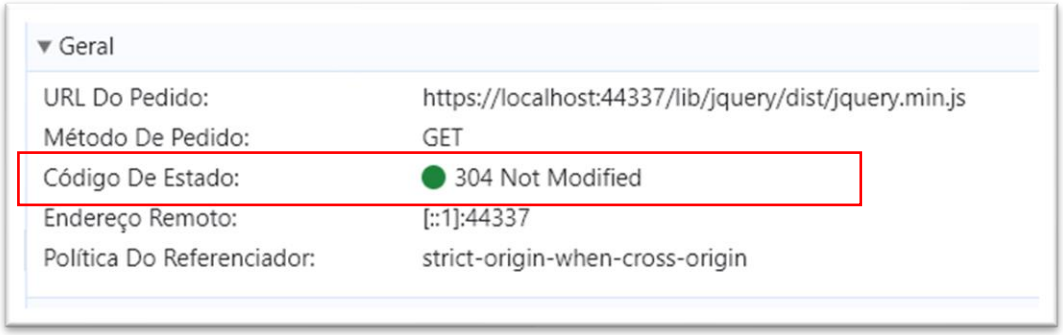
Change the **Program.cs** file by adding **StaticFileOptions** to the **UseStaticFiles** method according to the following code.

```
app.UseHttpsRedirection();
app.UseStaticFiles(
    new StaticFileOptions
    {
        OnPrepareResponse = ctx =>
        {
            ctx.Context.Response.Headers[HeaderNames.CacheControl] = "public, max-age=" + 20;
            ctx.Context.Response.Headers[HeaderNames.LastModified] = ctx.File.LastModified.ToString();
        }
    }
);
```

Test the application and verify (in any of the requests) that, after the cached content expires, the requests are made conditionally through the **If-Modified-Since** header and the responses are returned with the status code **304 (Not Modified)**.







Second step – Creating custom filter action

The action filters we intend to implement are based on the **ActionFilterAttribute** class which implements the **IActionFilter** and **IResultFilter** interfaces.
This class exposes four methods that can be rewritten:

OnActionExecuting(): Fires before the action is executed.
OnActionExecuted(): Fires after the action is executed.
OnResultExecuting(): Fires before the action result is executed.
OnResultExecuted(): Fires after the action result is execute

Rewriting these methods allows you to modify its default behaviour providing to the filter the desired behaviour.
The customized filters can be parameterized through public properties.

Create a new folder in the **Solution Explorer** named **Filters**.

Within this folder, create a new class, named **DailyMaintenanceFilter** and derived from the **ActionFilterAttribute** class.

This filter is defined to restrict access to a resource on a daily basis within a time period defined by the **From** and **To** parameters.

```
public class DailyMaintenanceFilter : ActionFilterAttribute
{
    3 references
    public int[] From { get; set; } // hours, minutes, seconds
    3 references
    public int[] To { get; set; } // hours, minutes, seconds

    TimeSpan _From, _To;

    0 references
    public override void OnActionExecuting(ActionExecutingContext context)
    {
        _From = new TimeSpan(From[0], From[1], From[2]);
        _To = new TimeSpan(To[0], To[1], To[2]);

        DateTime _justNow = DateTime.Now;
        TimeSpan _now = new TimeSpan(_justNow.Hour, _justNow.Minute, _justNow.Second);

        if ((_From <= _To && _now >= _From && _now <= _To) || (_From > _To && (_now > _From || _now < _To)))
        {
            context.Result = new RedirectToRouteResult(
                new RouteValueDictionary(
                    new { controller = "Security", action = "Maintenance" }));
        }
        else
            base.OnActionExecuting(context);
    }
}
```

Implement a new controller named **SecurityController**, with an action named **Maintenance**.

```
0 references
public class SecurityController : Controller
{
    0 references
    public IActionResult Maintenance()
    {
        HttpContext.Response.StatusCode = 405;
        return View();
    }
}
```

For this action, implement the following view.

```
@{
    ViewData["Title"] = "Maintenance";
}

<h1>Site Maintenance</h1>

<img src="~/images/maintenance.gif" width="500" height="500"/>

<p>This site it´s closed to daily maintenance.</p>
<p>We are sorry for the inconvenience. Please come back later.</p>
```

In the **wwwroot** folder of **Soluction Explorer**, create an **Images** folder.
Add a file in the created folder with an image that symbolizes maintenance (in the example: maintenance.gif).

Apply the filter created in the **HomeController**, parameterizing it for the time interval from 22h30m00s to 23h59m59s.

```
[DailyMaintenanceFilter(From = new[] {22,30,0}, To = new[] {23,59,59})]
3 references
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;

    0 references
    public HomeController(ILogger<HomeController> logger)
    {
        _logger = logger;
    }
```

Try the application.
If the access to the application is in an hourly time between the values defined in the filter, the result should be as follows.
Outside the indicated range, the application works as normal.