

The encoder safe lists can be customized to include Unicode ranges appropriate to the app during startup, in `Program.cs`:

For example, using the default configuration using a Razor HtmlHelper similar to the following:

HTML

```
<p>This link text is in Chinese: @Html.ActionLink("汉语/漢語", "Index")</p>
```

The preceding markup is rendered with Chinese text encoded:

HTML

```
<p>This link text is in Chinese: <a  
href="/">&#x6C49;&#x8BED;/&#x6F22;&#x8A9E;</a></p>
```

To widen the characters treated as safe by the encoder, insert the following line into `Program.cs`:

C#

```
builder.Services.AddSingleton<HtmlEncoder>(  
    HtmlEncoder.Create(allowedRanges: new[] { UnicodeRanges.BasicLatin,  
    UnicodeRanges.CjkUnifiedIdeographs }));
```

This example widens the safe list to include the Unicode Range `CjkUnifiedIdeographs`. The rendered output would now become

HTML

```
<p>This link text is in Chinese: <a href="/">汉语/漢語</a></p>
```

Safe list ranges are specified as Unicode code charts, not languages. The [Unicode standard](#) has a list of [code charts](#) you can use to find the chart containing your characters. Each encoder, Html, JavaScript and Url, must be configured separately.

ⓘ Note

Customization of the safe list only affects encoders sourced via DI. If you directly access an encoder via `System.Text.Encodings.Web.*Encoder.Default` then the default, Basic Latin only safelist will be used.

Where should encoding take place?

The general accepted practice is that encoding takes place at the point of output and encoded values should never be stored in a database. Encoding at the point of output allows you to change the use of data, for example, from HTML to a query string value. It also enables you to easily search your data without having to encode values before searching and allows you to take advantage of any changes or bug fixes made to encoders.

Validation as an XSS prevention technique

Validation can be a useful tool in limiting XSS attacks. For example, a numeric string containing only the characters 0-9 won't trigger an XSS attack. Validation becomes more complicated when accepting HTML in user input. Parsing HTML input is difficult, if not impossible. Markdown, coupled with a parser that strips embedded HTML, is a safer option for accepting rich input. Never rely on validation alone. Always encode untrusted input before output, no matter what validation or sanitization has been performed.

Enable Cross-Origin Requests (CORS) in ASP.NET Core

Article • 09/21/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

By [Rick Anderson](#) and [Kirk Larkin](#)

This article shows how Cross-Origin Resource Sharing ([CORS](#)) is enabled in an ASP.NET Core app.

Browser security prevents a web page from making requests to a different domain than the one that served the web page. This restriction is called the *same-origin policy*. The same-origin policy prevents a malicious site from reading sensitive data from another site. Sometimes, you might want to allow other sites to make cross-origin requests to your app. For more information, see the [Mozilla CORS article](#).

[Cross Origin Resource Sharing](#) (CORS):

- Is a W3C standard that allows a server to relax the same-origin policy.
- Is **not** a security feature, CORS relaxes security. An API is not safer by allowing CORS. For more information, see [How CORS works](#).
- Allows a server to explicitly allow some cross-origin requests while rejecting others.
- Is safer and more flexible than earlier techniques, such as [JSONP](#).

[View or download sample code](#) (how to download)

Same origin

Two URLs have the same origin if they have identical schemes, hosts, and ports ([RFC 6454](#)).

These two URLs have the same origin:

- `https://example.com/foo.html`

- `https://example.com/bar.html`

These URLs have different origins than the previous two URLs:

- `https://example.net`: Different domain
- `https://contoso.example.com/foo.html`: Different subdomain
- `http://example.com/foo.html`: Different scheme
- `https://example.com:9000/foo.html`: Different port

Enable CORS

There are three ways to enable CORS:

- In middleware using a [named policy](#) or [default policy](#).
- Using [endpoint routing](#).
- With the [\[EnableCors\]](#) attribute.

Using the [\[EnableCors\]](#) attribute with a named policy provides the finest control in limiting endpoints that support CORS.

Warning

[UseCors](#) must be called in the correct order. For more information, see [Middleware order](#). For example, [UseCors](#) must be called before [UseResponseCaching](#) when using [UseResponseCaching](#).

Each approach is detailed in the following sections.

CORS with named policy and middleware

CORS Middleware handles cross-origin requests. The following code applies a CORS policy to all the app's endpoints with the specified origins:

C#

```
var MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

var builder = WebApplication.CreateBuilder(args);
```

```

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: MyAllowSpecificOrigins,
        policy =>
        {
            policy.WithOrigins("http://example.com",
                               "http://www.contoso.com");
        });
});

// services.AddResponseCaching();

builder.Services.AddControllers();

var app = builder.Build();
app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseCors(MyAllowSpecificOrigins);

app.UseAuthorization();

app.MapControllers();

app.Run();

```

The preceding code:

- Sets the policy name to `_myAllowSpecificOrigins`. The policy name is arbitrary.
- Calls the `UseCors` extension method and specifies the `_myAllowSpecificOrigins` CORS policy. `UseCors` adds the CORS middleware. The call to `UseCors` must be placed after `UseRouting`, but before `UseAuthorization`. For more information, see [Middleware order](#).
- Calls `AddCors` with a [lambda expression](#). The lambda takes a `CorsPolicyBuilder` object. [Configuration options](#), such as `WithOrigins`, are described later in this article.
- Enables the `_myAllowSpecificOrigins` CORS policy for all controller endpoints. See [endpoint routing](#) to apply a CORS policy to specific endpoints.
- When using [Response Caching Middleware](#), call `UseCors` before `UseResponseCaching`.

With endpoint routing, the CORS middleware **must** be configured to execute between the calls to `UseRouting` and `UseEndpoints`.

The `AddCors` method call adds CORS services to the app's service container:

```

var MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: MyAllowSpecificOrigins,
        policy =>
        {
            policy.WithOrigins("http://example.com",
                               "http://www.contoso.com");
        });
});

// services.AddResponseCaching();

builder.Services.AddControllers();

var app = builder.Build();
app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseCors(MyAllowSpecificOrigins);

app.UseAuthorization();

app.MapControllers();

app.Run();

```

For more information, see [CORS policy options](#) in this document.

The [CorsPolicyBuilder](#) methods can be chained, as shown in the following code:

```

C#

var MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(MyAllowSpecificOrigins,
        policy =>
        {
            policy.WithOrigins("http://example.com",
                               "http://www.contoso.com")
                .AllowAnyHeader()
                .AllowAnyMethod();
        });
});

```

```
builder.Services.AddControllers();

var app = builder.Build();
app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseCors(MyAllowSpecificOrigins);

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Note: The specified URL must **not** contain a trailing slash (/). If the URL terminates with /, the comparison returns `false` and no header is returned.

UseCors and UseStaticFiles order

Typically, `UseStaticFiles` is called before `UseCors`. Apps that use JavaScript to retrieve static files cross site must call `UseCors` before `UseStaticFiles`.

CORS with default policy and middleware

The following highlighted code enables the default CORS policy:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddDefaultPolicy(
        policy =>
        {
            policy.WithOrigins("http://example.com",
                              "http://www.contoso.com");
        });
});

builder.Services.AddControllers();

var app = builder.Build();

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();
```

```
app.UseCors();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

The preceding code applies the default CORS policy to all controller endpoints.

Enable Cors with endpoint routing

With endpoint routing, CORS can be enabled on a per-endpoint basis using the [RequireCors](#) set of extension methods:

```
C#

var MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: MyAllowSpecificOrigins,
        policy =>
        {
            policy.WithOrigins("http://example.com",
                               "http://www.contoso.com");
        });
});

builder.Services.AddControllers();
builder.Services.AddRazorPages();

var app = builder.Build();

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseCors();

app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/echo",
        context => context.Response.WriteAsync("echo"))
        .RequireCors(MyAllowSpecificOrigins);
});
```



```

endpoints.MapControllers()
    .RequireCors(MyAllowSpecificOrigins);

endpoints.MapGet("/echo2",
    context => context.Response.WriteAsync("echo2"));

endpoints.MapRazorPages();
});

app.Run();

```

In the preceding code:

- `app.UseCors` enables the CORS middleware. Because a default policy hasn't been configured, `app.UseCors()` alone doesn't enable CORS.
- The `/echo` and controller endpoints allow cross-origin requests using the specified policy.
- The `/echo2` and Razor Pages endpoints do **not** allow cross-origin requests because no default policy was specified.

The [\[DisableCors\]](#) attribute does **not** disable CORS that has been enabled by endpoint routing with `RequireCors`.

See [Test CORS with \[EnableCors\] attribute and RequireCors method](#) for instructions on testing code similar to the preceding.

Enable CORS with attributes

Enabling CORS with the [\[EnableCors\]](#) attribute and applying a named policy to only those endpoints that require CORS provides the finest control.

The [\[EnableCors\]](#) attribute provides an alternative to applying CORS globally. The `[EnableCors]` attribute enables CORS for selected endpoints, rather than all endpoints:

- `[EnableCors]` specifies the default policy.
- `[EnableCors("{Policy String}")]` specifies a named policy.

The `[EnableCors]` attribute can be applied to:

- Razor Page `PageModel`
- Controller
- Controller action method

Different policies can be applied to controllers, page models, or action methods with the `[EnableCors]` attribute. When the `[EnableCors]` attribute is applied to a controller, page

model, or action method, and CORS is enabled in middleware, **both** policies are applied. We recommend against combining policies. Use the `[EnableCors]` attribute or middleware, not both in the same app.

The following code applies a different policy to each method:

```
C#

[Route("api/[controller]")]
[ApiController]
public class WidgetController : ControllerBase
{
    // GET api/values
    [EnableCors("AnotherPolicy")]
    [HttpGet]
    public ActionResult<IEnumerable<string>> Get()
    {
        return new string[] { "green widget", "red widget" };
    }

    // GET api/values/5
    [EnableCors("Policy1")]
    [HttpGet("{id}")]
    public ActionResult<string> Get(int id)
    {
        return id switch
        {
            1 => "green widget",
            2 => "red widget",
            _ => NotFound(),
        };
    }
}
```

The following code creates two CORS policies:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy("Policy1",
        policy =>
        {
            policy.WithOrigins("http://example.com",
                              "http://www.contoso.com");
        });

    options.AddPolicy("AnotherPolicy",
        policy =>
```

```

        {
            policy.WithOrigins("http://www.contoso.com")
                    .AllowAnyHeader()
                    .AllowAnyMethod();
        });
    });

builder.Services.AddControllers();

var app = builder.Build();

app.UseHttpsRedirection();

app.UseRouting();

app.UseCors();

app.UseAuthorization();

app.MapControllers();

app.Run();

```

For the finest control of limiting CORS requests:

- Use `[EnableCors("MyPolicy")]` with a named policy.
- Don't define a default policy.
- Don't use [endpoint routing](#).

The code in the next section meets the preceding list.

Disable CORS

The `[DisableCors]` attribute does **not** disable CORS that has been enabled by [endpoint routing](#).

The following code defines the CORS policy `"MyPolicy"`:

C#

```

var MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: "MyPolicy",
        policy =>
        {
            policy.WithOrigins("http://example.com",

```

```

        "http://www.contoso.com")
        .WithMethods("PUT", "DELETE", "GET");
    });
});

builder.Services.AddControllers();
builder.Services.AddRazorPages();

var app = builder.Build();

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseCors();

app.UseAuthorization();

app.UseEndpoints(endpoints => {
    endpoints.MapControllers();
    endpoints.MapRazorPages();
});

app.Run();

```

The following code disables CORS for the `GetValues2` action:

C#

```

[EnableCors("MyPolicy")]
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
    // GET api/values
    [HttpGet]
    public IActionResult Get() =>
        ControllerContext.MyDisplayRouteInfo();

    // GET api/values/5
    [HttpGet("{id}")]
    public IActionResult Get(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);

    // PUT api/values/5
    [HttpPut("{id}")]
    public IActionResult Put(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);

    // GET: api/values/GetValues2
    [DisableCors]
    [HttpGet("{action}")]

```

```
public IActionResult GetValues2() =>
    ControllerContext.MyDisplayRouteInfo();

}
```

The preceding code:

- Doesn't enable CORS with [endpoint routing](#).
- Doesn't define a [default CORS policy](#).
- Uses `[EnableCors("MyPolicy")]` to enable the `"MyPolicy"` CORS policy for the controller.
- Disables CORS for the `GetValues2` method.

See [Test CORS](#) for instructions on testing the preceding code.

CORS policy options

This section describes the various options that can be set in a CORS policy:

- [Set the allowed origins](#)
- [Set the allowed HTTP methods](#)
- [Set the allowed request headers](#)
- [Set the exposed response headers](#)
- [Credentials in cross-origin requests](#)
- [Set the preflight expiration time](#)

`AddPolicy` is called in `Program.cs`. For some options, it may be helpful to read the [How CORS works](#) section first.

Set the allowed origins

`AllowAnyOrigin`: Allows CORS requests from all origins with any scheme (`http` or `https`).

`AllowAnyOrigin` is insecure because *any website* can make cross-origin requests to the app.

ⓘ Note

Specifying `AllowAnyOrigin` and `AllowCredentials` is an insecure configuration and can result in cross-site request forgery. The CORS service returns an invalid CORS response when an app is configured with both methods.

`AllowAnyOrigin` affects preflight requests and the `Access-Control-Allow-Origin` header. For more information, see the [Preflight requests](#) section.

[SetIsOriginAllowedToAllowWildcardSubdomains](#): Sets the `IsOriginAllowed` property of the policy to be a function that allows origins to match a configured wildcard domain when evaluating if the origin is allowed.

C#

```
var MyAllowSpecificOrigins = "_MyAllowSubdomainPolicy";

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: MyAllowSpecificOrigins,
        policy =>
        {
            policy.WithOrigins("https://*.example.com")
                .SetIsOriginAllowedToAllowWildcardSubdomains();
        });
});

builder.Services.AddControllers();

var app = builder.Build();
```

Set the allowed HTTP methods

[AllowAnyMethod](#):

- Allows any HTTP method:
- Affects preflight requests and the `Access-Control-Allow-Methods` header. For more information, see the [Preflight requests](#) section.

Set the allowed request headers

To allow specific headers to be sent in a CORS request, called *author request headers*, call [WithHeaders](#) and specify the allowed headers:

C#

```
using Microsoft.Net.Http.Headers;

var MyAllowSpecificOrigins = "_MyAllowSubdomainPolicy";

var builder = WebApplication.CreateBuilder(args);
```

```
builder.Services.AddCors(options =>
{
    options.AddPolicy(name: MyAllowSpecificOrigins,
        policy =>
        {
            policy.WithOrigins("http://example.com")
                .WithHeaders(HeaderNames.ContentType, "x-custom-header");
        });
});

builder.Services.AddControllers();

var app = builder.Build();
```

To allow all author request headers, call [AllowAnyHeader](#):

```
C#

var MyAllowSpecificOrigins = "_MyAllowSubdomainPolicy";

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: MyAllowSpecificOrigins,
        policy =>
        {
            policy.WithOrigins("https://*.example.com")
                .AllowAnyHeader();
        });
});

builder.Services.AddControllers();

var app = builder.Build();
```

`AllowAnyHeader` affects preflight requests and the [Access-Control-Request-Headers](#) header. For more information, see the [Preflight requests](#) section.

A CORS Middleware policy match to specific headers specified by `WithHeaders` is only possible when the headers sent in `Access-Control-Request-Headers` exactly match the headers stated in `WithHeaders`.

For instance, consider an app configured as follows:

```
C#

app.UseCors(policy => policy.WithHeaders(HeaderNames.CacheControl));
```

CORS Middleware declines a preflight request with the following request header because `Content-Language` ([HeaderNames.ContentLanguage](#)) isn't listed in `WithHeaders`:

```
Access-Control-Request-Headers: Cache-Control, Content-Language
```

The app returns a *200 OK* response but doesn't send the CORS headers back. Therefore, the browser doesn't attempt the cross-origin request.

Set the exposed response headers

By default, the browser doesn't expose all of the response headers to the app. For more information, see [W3C Cross-Origin Resource Sharing \(Terminology\): Simple Response Header](#).

The response headers that are available by default are:

- `Cache-Control`
- `Content-Language`
- `Content-Type`
- `Expires`
- `Last-Modified`
- `Pragma`

The CORS specification calls these headers *simple response headers*. To make other headers available to the app, call [WithExposedHeaders](#):

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy("MyExposeResponseHeadersPolicy",
        policy =>
        {
            policy.WithOrigins("https://*.example.com")
                .WithExposedHeaders("x-custom-header");
        });
});

builder.Services.AddControllers();

var app = builder.Build();
```


Credentials in cross-origin requests

Credentials require special handling in a CORS request. By default, the browser doesn't send credentials with a cross-origin request. Credentials include cookies and HTTP authentication schemes. To send credentials with a cross-origin request, the client must set `XMLHttpRequest.withCredentials` to `true`.

Using `XMLHttpRequest` directly:

JavaScript

```
var xhr = new XMLHttpRequest();
xhr.open('get', 'https://www.example.com/api/test');
xhr.withCredentials = true;
```

Using jQuery:

JavaScript

```
$.ajax({
  type: 'get',
  url: 'https://www.example.com/api/test',
  xhrFields: {
    withCredentials: true
  }
});
```

Using the [Fetch API](#):

JavaScript

```
fetch('https://www.example.com/api/test', {
  credentials: 'include'
});
```

The server must allow the credentials. To allow cross-origin credentials, call [AllowCredentials](#):

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy("MyMyAllowCredentialsPolicy",
        policy =>
        {
```

```
        policy.WithOrigins("http://example.com")
            .AllowCredentials();
    });
});

builder.Services.AddControllers();

var app = builder.Build();
```

The HTTP response includes an `Access-Control-Allow-Credentials` header, which tells the browser that the server allows credentials for a cross-origin request.

If the browser sends credentials but the response doesn't include a valid `Access-Control-Allow-Credentials` header, the browser doesn't expose the response to the app, and the cross-origin request fails.

Allowing cross-origin credentials is a security risk. A website at another domain can send a signed-in user's credentials to the app on the user's behalf without the user's knowledge.

The CORS specification also states that setting origins to `"*"` (all origins) is invalid if the `Access-Control-Allow-Credentials` header is present.

Preflight requests

For some CORS requests, the browser sends an additional [OPTIONS](#) request before making the actual request. This request is called a [preflight request](#). The browser can skip the preflight request if **all** the following conditions are true:

- The request method is GET, HEAD, or POST.
- The app doesn't set request headers other than `Accept`, `Accept-Language`, `Content-Language`, `Content-Type`, or `Last-Event-ID`.
- The `Content-Type` header, if set, has one of the following values:
 - `application/x-www-form-urlencoded`
 - `multipart/form-data`
 - `text/plain`

The rule on request headers set for the client request applies to headers that the app sets by calling `setRequestHeader` on the `XMLHttpRequest` object. The CORS specification calls these headers *author request headers*. The rule doesn't apply to headers the browser can set, such as `User-Agent`, `Host`, or `Content-Length`.

ⓘ Note

This article contains URLs created by deploying the [sample code](#) to two Azure web sites, <https://cors3.azurewebsites.net> and <https://cors.azurewebsites.net>.

The following is an example response similar to the preflight request made from the **[Put test]** button in the [Test CORS](#) section of this document.

```
General:
Request URL: https://cors3.azurewebsites.net/api/values/5
Request Method: OPTIONS
Status Code: 204 No Content

Response Headers:
Access-Control-Allow-Methods: PUT,DELETE,GET
Access-Control-Allow-Origin: https://cors1.azurewebsites.net
Server: Microsoft-IIS/10.0
Set-Cookie:
ARRAffinity=8f8...8;Path=/;HttpOnly;Domain=cors1.azurewebsites.net
Vary: Origin


Request Headers:
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Access-Control-Request-Method: PUT
Connection: keep-alive
Host: cors3.azurewebsites.net
Origin: https://cors1.azurewebsites.net
Referer: https://cors1.azurewebsites.net/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: cross-site
User-Agent: Mozilla/5.0
```

The preflight request uses the [HTTP OPTIONS](#) method. It may include the following headers:

- [Access-Control-Request-Method](#): The HTTP method that will be used for the actual request.
- [Access-Control-Request-Headers](#): A list of request headers that the app sets on the actual request. As stated earlier, this doesn't include headers that the browser sets, such as `User-Agent`.

If the preflight request is denied, the app returns a `200 OK` response but doesn't set the CORS headers. Therefore, the browser doesn't attempt the cross-origin request. For an example of a denied preflight request, see the [Test CORS](#) section of this document.

Using the F12 tools, the console app shows an error similar to one of the following, depending on the browser:

- Firefox: Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at `https://cors1.azurewebsites.net/api/ToDoItems1/MyDelete2/5`. (Reason: CORS request did not succeed). [Learn More](#) 
- Chromium based: Access to fetch at 'https://cors1.azurewebsites.net/api/ToDoItems1/MyDelete2/5' from origin 'https://cors3.azurewebsites.net' has been blocked by CORS policy: Response to preflight request doesn't pass access control check: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.

To allow specific headers, call [WithHeaders](#):

C#

```
using Microsoft.Net.Http.Headers;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy("MyAllowHeadersPolicy",
        policy =>
        {
            policy.WithOrigins("http://example.com")
                .WithHeaders(HeaderNames.ContentType, "x-custom-header");
        });
});

builder.Services.AddControllers();

var app = builder.Build();
```

To allow all author request headers, call [AllowAnyHeader](#):

C#

```
using Microsoft.Net.Http.Headers;
```

```

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy("MyAllowAllHeadersPolicy",
        policy =>
        {
            policy.WithOrigins("https://*.example.com")
                .AllowAnyHeader();
        });
});

builder.Services.AddControllers();

var app = builder.Build();

```

Browsers aren't consistent in how they set `Access-Control-Request-Headers`. If either:

- Headers are set to anything other than `"*"`
- `AllowAnyHeader` is called: Include at least `Accept`, `Content-Type`, and `Origin`, plus any custom headers that you want to support.

Automatic preflight request code

When the CORS policy is applied either:

- Globally by calling `app.UseCors` in `Program.cs`.
- Using the `[EnableCors]` attribute.

ASP.NET Core responds to the preflight OPTIONS request.

The [Test CORS](#) section of this document demonstrates this behavior.

[HttpOptions] attribute for preflight requests

When CORS is enabled with the appropriate policy, ASP.NET Core generally responds to CORS preflight requests automatically.

The following code uses the `[HttpOptions]` attribute to create endpoints for OPTIONS requests:

```

C#

[Route("api/[controller]")]
[ApiController]
public class TodoItems2Controller : ControllerBase
{

```

```

// OPTIONS: api/ToDoItems2/5
[HttpOptions("{id}")]
public IActionResult PreflightRoute(int id)
{
    return NoContent();
}

// OPTIONS: api/ToDoItems2
[HttpOptions]
public IActionResult PreflightRoute()
{
    return NoContent();
}

[HttpPut("{id}")]
public IActionResult PutToDoItem(int id)
{
    if (id < 1)
    {
        return BadRequest();
    }

    return ControllerContext.MyDisplayRouteInfo(id);
}

```

See [Test CORS with \[EnableCors\] attribute and RequireCors method](#) for instructions on testing the preceding code.

Set the preflight expiration time

The `Access-Control-Max-Age` header specifies how long the response to the preflight request can be cached. To set this header, call [SetPreflightMaxAge](#):

C#

```

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy("MySetPreflightExpirationPolicy",
        policy =>
        {
            policy.WithOrigins("http://example.com")
                .SetPreflightMaxAge(TimeSpan.FromSeconds(2520));
        });
});

builder.Services.AddControllers();

var app = builder.Build();

```

Enable CORS on an endpoint

How CORS works

This section describes what happens in a [CORS](#) request at the level of the HTTP messages.

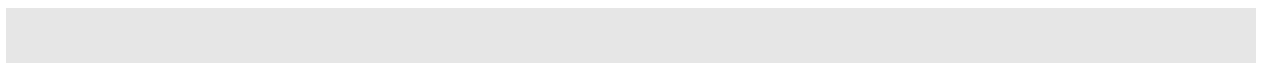
- CORS is **not** a security feature. CORS is a W3C standard that allows a server to relax the same-origin policy.
 - For example, a malicious actor could use [Cross-Site Scripting \(XSS\)](#) against your site and execute a cross-site request to their CORS enabled site to steal information.
- An API isn't safer by allowing CORS.
 - It's up to the client (browser) to enforce CORS. The server executes the request and returns the response, it's the client that returns an error and blocks the response. For example, any of the following tools will display the server response:
 - [Fiddler](#)
 - [.NET HttpClient](#)
 - A web browser by entering the URL in the address bar.
- It's a way for a server to allow browsers to execute a cross-origin [XHR](#) or [Fetch API](#) request that otherwise would be forbidden.
 - Browsers without CORS can't do cross-origin requests. Before CORS, [JSONP](#) was used to circumvent this restriction. JSONP doesn't use XHR, it uses the `<script>` tag to receive the response. Scripts are allowed to be loaded cross-origin.

The [CORS specification](#) introduced several new HTTP headers that enable cross-origin requests. If a browser supports CORS, it sets these headers automatically for cross-origin requests. Custom JavaScript code isn't required to enable CORS.

The following is an example of a cross-origin request from the **Values** test button to `https://cors1.azurewebsites.net/api/values`. The `Origin` header:

- Provides the domain of the site that's making the request.
- Is required and must be different from the host.

General headers



```
Request URL: https://cors1.azurewebsites.net/api/values
Request Method: GET
Status Code: 200 OK
```

Response headers

```
Content-Encoding: gzip
Content-Type: text/plain; charset=utf-8
Server: Microsoft-IIS/10.0
Set-Cookie: ARRAffinity=8f...;Path=/;HttpOnly;Domain=cors1.azurewebsites.net
Transfer-Encoding: chunked
Vary: Accept-Encoding
X-Powered-By: ASP.NET
```

Request headers

```
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Connection: keep-alive
Host: cors1.azurewebsites.net
Origin: https://cors3.azurewebsites.net
Referer: https://cors3.azurewebsites.net/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: cross-site
User-Agent: Mozilla/5.0 ...
```

In `OPTIONS` requests, the server sets the **Response headers** `Access-Control-Allow-Origin: {allowed origin}` header in the response. For example, in the [sample code](#), the `Delete [EnableCors]` button `OPTIONS` request contains the following headers:

General headers

```
Request URL: https://cors3.azurewebsites.net/api/ToDoItems2/MyDelete2/5
Request Method: OPTIONS
Status Code: 204 No Content
```

Response headers


```
Access-Control-Allow-Headers: Content-Type,x-custom-header
Access-Control-Allow-Methods: PUT,DELETE,GET,OPTIONS
Access-Control-Allow-Origin: https://cors1.azurewebsites.net
Server: Microsoft-IIS/10.0
Set-Cookie: ARRAffinity=8f...;Path=/;HttpOnly;Domain=cors3.azurewebsites.net
Vary: Origin
X-Powered-By: ASP.NET
```

Request headers

```
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Access-Control-Request-Headers: content-type
Access-Control-Request-Method: DELETE
Connection: keep-alive
Host: cors3.azurewebsites.net
Origin: https://cors1.azurewebsites.net
Referer: https://cors1.azurewebsites.net/test?number=2
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: cross-site
User-Agent: Mozilla/5.0
```

In the preceding **Response headers**, the server sets the [Access-Control-Allow-Origin](#) [↗](#) header in the response. The `https://cors1.azurewebsites.net` value of this header matches the `Origin` header from the request.

If [AllowAnyOrigin](#) is called, the `Access-Control-Allow-Origin: *`, the wildcard value, is returned. `AllowAnyOrigin` allows any origin.

If the response doesn't include the `Access-Control-Allow-Origin` header, the cross-origin request fails. Specifically, the browser disallows the request. Even if the server returns a successful response, the browser doesn't make the response available to the client app.

HTTP redirection to HTTPS causes ERR_INVALID_REDIRECT on the CORS preflight request

Requests to an endpoint using HTTP that are redirected to HTTPS by [UseHttpsRedirection](#) fail with `ERR_INVALID_REDIRECT` on the CORS preflight request.

API projects can reject HTTP requests rather than use `UseHttpsRedirection` to redirect requests to HTTPS.

CORS in IIS

When deploying to IIS, CORS has to run before Windows Authentication if the server isn't configured to allow anonymous access. To support this scenario, the [IIS CORS module](#) needs to be installed and configured for the app.

Test CORS

The [sample download](#) has code to test CORS. See [how to download](#). The sample is an API project with Razor Pages added:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: "MyPolicy",
        policy =>
        {
            policy.WithOrigins("http://example.com",
                "http://www.contoso.com",
                "https://cors1.azurewebsites.net",
                "https://cors3.azurewebsites.net",
                "https://localhost:44398",
                "https://localhost:5001")
                .WithMethods("PUT", "DELETE", "GET");
        });
});

builder.Services.AddControllers();
builder.Services.AddRazorPages();

var app = builder.Build();

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseCors();

app.UseAuthorization();

app.MapControllers();
app.MapRazorPages();
```

```
app.Run();
```

⚠ Warning

`WithOrigins("https://localhost:<port>");` should only be used for testing a sample app similar to the [download sample code](#) ↗.

The following `ValuesController` provides the endpoints for testing:

C#

```
[EnableCors("MyPolicy")]
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
    // GET api/values
    [HttpGet]
    public IActionResult Get() =>
        ControllerContext.MyDisplayRouteInfo();

    // GET api/values/5
    [HttpGet("{id}")]
    public IActionResult Get(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);

    // PUT api/values/5
    [HttpPut("{id}")]
    public IActionResult Put(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);

    // GET: api/values/GetValues2
    [DisableCors]
    [HttpGet("{action}")]
    public IActionResult GetValues2() =>
        ControllerContext.MyDisplayRouteInfo();
}
```

`MyDisplayRouteInfo` ↗ is provided by the [Rick.Docs.Samples.RouteInfo](#) ↗ NuGet package and displays route information.

Test the preceding sample code by using one of the following approaches:

- Run the sample with `dotnet run` using the default URL of `https://localhost:5001`.

- Run the sample from Visual Studio with the port set to 44398 for a URL of `https://localhost:44398`.

Using a browser with the F12 tools:

- Select the **Values** button and review the headers in the **Network** tab.
- Select the **PUT test** button. See [Display OPTIONS requests](#) for instructions on displaying the OPTIONS request. The **PUT test** creates two requests, an OPTIONS preflight request and the PUT request.
- Select the **GetValues2 [DisableCors]** button to trigger a failed CORS request. As mentioned in the document, the response returns 200 success, but the CORS request is not made. Select the **Console** tab to see the CORS error. Depending on the browser, an error similar to the following is displayed:

Access to fetch at '`https://cors1.azurewebsites.net/api/values/GetValues2`' from origin '`https://cors3.azurewebsites.net`' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.

CORS-enabled endpoints can be tested with a tool, such as [curl](#) or [Fiddler](#). When using a tool, the origin of the request specified by the `Origin` header must differ from the host receiving the request. If the request isn't *cross-origin* based on the value of the `Origin` header:

- There's no need for CORS Middleware to process the request.
- CORS headers aren't returned in the response.

The following command uses `curl` to issue an OPTIONS request with information:

Bash

```
curl -X OPTIONS https://cors3.azurewebsites.net/api/ToDoItems2/5 -i
```

Test CORS with [EnableCors] attribute and RequireCors method

Consider the following code which uses [endpoint routing](#) to enable CORS on a per-endpoint basis using `RequireCors`:

C#

```

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: "MyPolicy",
        policy =>
        {
            policy.WithOrigins("http://example.com",
                "http://www.contoso.com",
                "https://cors1.azurewebsites.net",
                "https://cors3.azurewebsites.net",
                "https://localhost:44398",
                "https://localhost:5001")
                .WithMethods("PUT", "DELETE", "GET");
        });
});

builder.Services.AddControllers();
builder.Services.AddRazorPages();

var app = builder.Build();

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseCors();

app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/echo",
        context => context.Response.WriteAsync("echo"))
        .RequireCors("MyPolicy");

    endpoints.MapControllers();
    endpoints.MapRazorPages();
});

app.Run();

```

Notice that only the `/echo` endpoint is using the `RequireCors` to allow cross-origin requests using the specified policy. The controllers below enable CORS using `[EnableCors]` attribute.

The following `TodoItems1Controller` provides endpoints for testing:

C#

```

[Route("api/[controller]")]
[ApiController]
public class TodoItems1Controller : ControllerBase
{
    // PUT: api/TodoItems1/5
    [HttpPut("{id}")]
    public IActionResult PutTodoItem(int id) {
        if (id < 1) {
            return Content($"ID = {id}");
        }

        return ControllerContext.MyDisplayRouteInfo(id);
    }

    // Delete: api/TodoItems1/5
    [HttpDelete("{id}")]
    public IActionResult MyDelete(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);

    // GET: api/TodoItems1
    [HttpGet]
    public IActionResult GetTodoItems() =>
        ControllerContext.MyDisplayRouteInfo();

    [EnableCors("MyPolicy")]
    [HttpGet("{action}")]
    public IActionResult GetTodoItems2() =>
        ControllerContext.MyDisplayRouteInfo();

    // Delete: api/TodoItems1/MyDelete2/5
    [EnableCors("MyPolicy")]
    [HttpDelete("{action}/{id}")]
    public IActionResult MyDelete2(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);
}

```

The **Delete** [EnableCors] and **GET** [EnableCors] buttons succeed, because the endpoints have [EnableCors] and respond to preflight requests. The other endpoints fails. The **GET** button fails, because the [JavaScript](#) sends:

JavaScript

```

headers: {
    "Content-Type": "x-custom-header"
},

```

The following `TodoItems2Controller` provides similar endpoints, but includes explicit code to respond to OPTIONS requests:

C#

```

[Route("api/[controller]")]
[ApiController]
public class TodoItems2Controller : ControllerBase
{
    // OPTIONS: api/TodoItems2/5
    [HttpOptions("{id}")]
    public IActionResult PreflightRoute(int id)
    {
        return NoContent();
    }

    // OPTIONS: api/TodoItems2
    [HttpOptions]
    public IActionResult PreflightRoute()
    {
        return NoContent();
    }

    [HttpPut("{id}")]
    public IActionResult PutTodoItem(int id)
    {
        if (id < 1)
        {
            return BadRequest();
        }

        return ControllerContext.MyDisplayRouteInfo(id);
    }

    // [EnableCors] // Not needed as OPTIONS path provided.
    [HttpDelete("{id}")]
    public IActionResult MyDelete(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);

    // [EnableCors] // Warning ASP0023 Route '{id}' conflicts with another
    // action route.
    // An HTTP request that matches multiple routes results
    // in an ambiguous
    // match error.
    [EnableCors("MyPolicy")] // Required for this path.
    [HttpGet]
    public IActionResult GetTodoItems() =>
        ControllerContext.MyDisplayRouteInfo();

    [HttpGet("{action}")]
    public IActionResult GetTodoItems2() =>
        ControllerContext.MyDisplayRouteInfo();

    [EnableCors("MyPolicy")] // Required for this path.
    [HttpDelete("{action}/{id}")]
    public IActionResult MyDelete2(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);
}

```

The preceding code can be tested by deploying the sample to Azure. In the **Controller** drop down list, select **Preflight** and then **Set Controller**. All the CORS calls to the `TodoItems2Controller` endpoints succeed.

Additional resources

- [Cross-Origin Resource Sharing \(CORS\)](#) 
- [IIS CORS module Configuration Reference](#)



Share authentication cookies among ASP.NET apps

Article • 06/17/2024

By [Rick Anderson](#) 

Websites often consist of individual web apps working together. To provide a single sign-on (SSO) experience, web apps within a site must share authentication cookies. To support this scenario, the data protection stack allows sharing Katana cookie authentication and ASP.NET Core cookie authentication tickets.

In the examples that follow:

- The authentication cookie name is set to a common value of `.AspNet.SharedCookie`.
- The `AuthenticationType` is set to `Identity.Application` either explicitly or by default.
- A common app name, `SharedCookieApp`, is used to enable the data protection system to share data protection keys.
- `Identity.Application` is used as the authentication scheme. Whatever scheme is used, it must be used consistently *within and across* the shared cookie apps either as the default scheme or by explicitly setting it. The scheme is used when encrypting and decrypting cookies, so a consistent scheme must be used across apps.
- A common [data protection key](#) storage location is used.
 - In ASP.NET Core apps, [PersistKeysToFileSystem](#) is used to set the key storage location.
 - In .NET Framework apps, Cookie Authentication Middleware uses an implementation of [DataProtectionProvider](#). `DataProtectionProvider` provides data protection services for the encryption and decryption of authentication cookie payload data. The `DataProtectionProvider` instance is isolated from the data protection system used by other parts of the app. [DataProtectionProvider.Create\(System.IO.DirectoryInfo, Action<IDataProtectionBuilder>\)](#) accepts a [DirectoryInfo](#) to specify the location for data protection key storage.
- `DataProtectionProvider` requires the [Microsoft.AspNetCore.DataProtection.Extensions](#)  NuGet package:
 - In .NET Framework apps, add a package reference to [Microsoft.AspNetCore.DataProtection.Extensions](#) .

- [SetApplicationName](#) sets the common app name.

Share authentication cookies with ASP.NET Core Identity

When using ASP.NET Core Identity:

- Data protection keys and the app name must be shared among apps. A common key storage location is provided to the [PersistKeysToFileSystem](#) method in the following examples. Use [SetApplicationName](#) to configure a common shared app name (`SharedCookieApp` in the following examples). For more information, see [Configure ASP.NET Core Data Protection](#).
- Use the [ConfigureApplicationCookie](#) extension method to set up the data protection service for cookies.
- The default authentication type is `Identity.Application`.

In `Program.cs`:

```
C#  
  
using Microsoft.AspNetCore.DataProtection;  
  
var builder = WebApplication.CreateBuilder(args);  
  
builder.Services.AddRazorPages();  
  
builder.Services.AddDataProtection()  
    .PersistKeysToFileSystem(new DirectoryInfo(@"c:\PATH TO COMMON KEY RING  
FOLDER"))  
    .SetApplicationName("SharedCookieApp");  
  
builder.Services.ConfigureApplicationCookie(options => {  
    options.Cookie.Name = ".AspNet.SharedCookie";  
});  
  
var app = builder.Build();
```

Note: The preceding instructions don't work with `ITicketStore` (`CookieAuthenticationOptions.SessionStore`). For more information, see [this GitHub issue](#).

For security reasons, authentication cookies are not compressed in ASP.NET Core. When using authentication cookies, developers should minimize the number of claim information included to just that necessary for their needs.

Share authentication cookies without ASP.NET Core Identity

When using cookies directly without ASP.NET Core Identity, configure data protection and authentication. In the following example, the authentication type is set to

`Identity.Application`:

C#

```
using Microsoft.AspNetCore.DataProtection;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddDataProtection()
    .PersistKeysToFileSystem(new DirectoryInfo(@"c:\PATH TO COMMON KEY RING FOLDER"))
    .SetApplicationName("SharedCookieApp");

builder.Services.AddAuthentication("Identity.Application")
    .AddCookie("Identity.Application", options =>
    {
        options.Cookie.Name = ".AspNet.SharedCookie";
    });

var app = builder.Build();
```

For security reasons, authentication cookies are not compressed in ASP.NET Core. When using authentication cookies, developers should minimize the number of claim information included to just that necessary for their needs.

Share cookies across different base paths

An authentication cookie uses the `HttpRequest.PathBase` as its default `Cookie.Path`. If the app's cookie must be shared across different base paths, `Path` must be overridden:

C#

```
using Microsoft.AspNetCore.DataProtection;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
```

```
builder.Services.AddDataProtection()
    .PersistKeysToFileSystem(new DirectoryInfo(@"c:\PATH TO COMMON KEY RING
FOLDER"))
    .SetApplicationName("SharedCookieApp");

builder.Services.ConfigureApplicationCookie(options => {
    options.Cookie.Name = ".AspNet.SharedCookie";
    options.Cookie.Path = "/";
});
var app = builder.Build();
```

Share cookies across subdomains

When hosting apps that share cookies across subdomains, specify a common domain in the [Cookie.Domain](#) property. To share cookies across apps at `contoso.com`, such as `first_subdomain.contoso.com` and `second_subdomain.contoso.com`, specify the `Cookie.Domain` as `.contoso.com`:

C#

```
options.Cookie.Domain = ".contoso.com";
```

Encrypt data protection keys at rest

For production deployments, configure the `DataProtectionProvider` to encrypt keys at rest with DPAPI or an X509Certificate. For more information, see [Key encryption at rest in Windows and Azure using ASP.NET Core](#). In the following example, a certificate thumbprint is provided to [ProtectKeysWithCertificate](#):

C#

```
using Microsoft.AspNetCore.DataProtection;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddDataProtection()
    .ProtectKeysWithCertificate("{CERTIFICATE THUMBPRINT}");
```

Use a common user database

When apps use the same Identity schema (same version of Identity), confirm that the Identity system for each app is pointed at the same user database. Otherwise, the identity system produces failures at runtime when it attempts to match the information in the authentication cookie against the information in its database.

When the Identity schema is different among apps, usually because apps are using different Identity versions, sharing a common database based on the latest version of Identity isn't possible without remapping and adding columns in other app's Identity schemas. It's often more efficient to upgrade the other apps to use the latest Identity version so that a common database can be shared by the apps.

Application name change

In .NET 6, [WebApplicationBuilder](#) normalizes the content root path to end with a [DirectorySeparatorChar](#). Most apps migrating from [HostBuilder](#) or [WebHostBuilder](#) won't have the same app name because they aren't normalized. For more information, see [SetApplicationName](#)

Share authentication cookies between ASP.NET 4.x and ASP.NET Core apps

ASP.NET 4.x apps that use Microsoft.Owin Cookie Authentication Middleware can be configured to generate authentication cookies that are compatible with the ASP.NET Core Cookie Authentication Middleware. This can be useful if a web application consists of both ASP.NET 4.x apps and ASP.NET Core apps that must share a single sign-on experience. A specific example of such a scenario is [incrementally migrating](#) a web app from ASP.NET to ASP.NET Core. In such scenarios, it's common for some parts of an app to be served by the original ASP.NET app while others are served by the new ASP.NET Core app. Users should only have to sign in once, though. This can be accomplished by either of the following approaches:

- Using the System.Web adapters' [remote authentication](#) feature, which uses the ASP.NET app to sign users in.
- Configuring the ASP.NET app to use Microsoft.Owin Cookie Authentication Middleware so that authentication cookies are shared with the ASP.NET Core app.

To configure ASP.NET Microsoft.Owin Cookie Authentication Middleware to share cookies with an ASP.NET Core app, follow the preceding instructions to configure the ASP.NET Core app to use a specific cookie name, app name, and to persist data protection keys to a well-known location. See [Configure ASP.NET Core Data Protection](#) for more information on persisting data protection keys.

In the ASP.NET app, install the [Microsoft.Owin.Security.Interop](#) package.

Update the `UseCookieAuthentication` call in `Startup.Auth.cs` to configure an `AspNetTicketDataFormat` to match the ASP.NET Core app's settings:

C#

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    LoginPath = new PathString("/Account/Login"),
    Provider = new CookieAuthenticationProvider
    {
        OnValidateIdentity =
SecurityStampValidator.OnValidateIdentity<ApplicationUserManager,
ApplicationUser>(
            validateInterval: TimeSpan.FromMinutes(30),
            regenerateIdentity: (manager, user) =>
user.GenerateUserIdentityAsync(manager))
    },

    // Settings to configure shared cookie with ASP.NET Core app
    CookieName = ".AspNet.ApplicationCookie",
    AuthenticationType = "Identity.Application",
    TicketDataFormat = new AspNetTicketDataFormat(
        new DataProtectorShim(
            DataProtectionProvider.Create(new DirectoryInfo(@"c:\PATH TO
COMMON KEY RING FOLDER")),
            builder => builder.SetApplicationName("SharedCookieApp"))
        .CreateProtector(

"Microsoft.AspNetCore.Authentication.Cookies.CookieAuthenticationMiddleware"
,
            // Must match the Scheme name used in the ASP.NET Core app,
i.e. IdentityConstants.ApplicationScheme
            "Identity.Application",
            "v2"))),
    CookieManager = new ChunkingCookieManager()
});
```

Important items configured here include:

- The cookie name is set to the same name as in the ASP.NET Core app.
- A data protection provider is created using the same key ring path. Note that in these examples, data protection keys are stored on disk but other data protection providers can be used. For example, Redis or Azure Blob Storage can be used for data protection providers as long as the configuration matches between the apps. See [Configure ASP.NET Core Data Protection](#) for more information on persisting data protection keys.

- The app name is set to be the same as the app name used in the ASP.NET Core app.
- The authentication type is set to the name of the authentication scheme in the ASP.NET Core app.
- `System.Web.Helpers.AntiForgeryConfig.UniqueClaimTypeIdentifier` is set to a claim from the ASP.NET Core identity that will be unique to a user.

Because the authentication type was changed to match the authentication scheme of the ASP.NET Core app, it's also necessary to update how the ASP.NET app generates new identities to use that same name. This is typically done in

`Models/IdentityModels.cs`:

C#

```
public class ApplicationUser : IdentityUser
{
    public async Task<ClaimsIdentity>
GenerateUserIdentityAsync(UserManager<ApplicationUser> manager)
    {
        // Note the authenticationType must match the one defined in
CookieAuthenticationOptions.AuthenticationType
        var userIdentity = await manager.CreateIdentityAsync(this,
"Identity.Application");

        // Add custom user claims here
        return userIdentity;
    }
}
```

With these changes, the ASP.NET and ASP.NET Core apps are able to use the same authentication cookies so that users signing in or out of one app are reflected in the other app.

Note that because there are differences between ASP.NET Identity and ASP.NET Core Identity's database schemas, it is recommended that users only *sign-in* using one of the apps - either the ASP.NET or ASP.NET Core app. Once users are signed in, the steps documented in this section will allow the authentication cookie to be *used* by either app and both apps should be able to log users out.

Additional resources

- [Host ASP.NET Core in a web farm](#)
- [A primer on OWIN cookie authentication middleware for the ASP.NET developer](#) by Brock Allen
- [OWIN Authentication Middleware Architecture](#) by Brock Allen

- [Posts from the 'OWIN / Katana' Category](#) by [Brock Allen](#)

Work with SameSite cookies in ASP.NET Core

Article • 06/17/2024

By [Rick Anderson](#) [↗](#)

SameSite is an [IETF](#) [↗](#) draft standard designed to provide some protection against cross-site request forgery (CSRF) attacks. Originally drafted in [2016](#) [↗](#), the draft standard was updated in [2019](#) [↗](#). The updated standard is not backward compatible with the previous standard, with the following being the most noticeable differences:

- Cookies without SameSite header are treated as `SameSite=Lax` by default.
- `SameSite=None` must be used to allow cross-site cookie use.
- Cookies that assert `SameSite=None` must also be marked as `Secure`.
- Applications that use `<iframe>` [↗](#) may experience issues with `sameSite=Lax` or `sameSite=Strict` cookies because `<iframe>` is treated as cross-site scenarios.
- The value `SameSite=None` is not allowed by the [2016 standard](#) [↗](#) and causes some implementations to treat such cookies as `SameSite=Strict`. See [Supporting older browsers](#) in this document.

The `SameSite=Lax` setting works for most application cookies. Some forms of authentication like [OpenID Connect](#) [↗](#) (OIDC) and [WS-Federation](#) [↗](#) default to POST based redirects. The POST based redirects trigger the SameSite browser protections, so SameSite is disabled for these components. Most [OAuth](#) [↗](#) logins are not affected due to differences in how the request flows.

Each ASP.NET Core component that emits cookies needs to decide if SameSite is appropriate.

SameSite and Identity


ASP.NET Core [Identity](#) is largely unaffected by [SameSite cookies](#) except for advanced scenarios like `IFrames` or `OpenIdConnect` integration.

When using `Identity`, do **not** add any cookie providers or call `services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)`, `Identity` takes care of that.

SameSite test sample code

The following sample can be downloaded and tested:

 Expand table

Sample	Document
.NET Core Razor Pages 	ASP.NET Core 3.1 Razor Pages SameSite cookie sample

.NET Core support for the sameSite attribute

.NET Core supports the 2019 draft standard for SameSite. Developers are able to programmatically control the value of the sameSite attribute using the `HttpContext.Response.Cookies.Append` property. Setting the `SameSite` property to `Strict`, `Lax`, or `None` results in those values being written on the network with the cookie. Setting to `SameSiteMode.Unspecified` indicates no sameSite should be sent with the cookie.

C#

```
var cookieOptions = new CookieOptions
{
    // Set the secure flag, which Chrome's changes will require for
    SameSite none.
    // Note this will also require you to be running on HTTPS.
    Secure = true,

    // Set the cookie to HTTP only which is good practice unless you
    really do need
    // to access it client side in scripts.
    HttpOnly = true,

    // Add the SameSite attribute, this will emit the attribute with a
    value of none.
    SameSite = SameSiteMode.None

    // The client should follow its default cookie policy.
    // SameSite = SameSiteMode.Unspecified
};

// Add the cookie to the response cookie collection
Response.Cookies.Append("MyCookie", "cookieValue", cookieOptions);
}
```

API usage with SameSite

`HttpContext.Response.Cookies.Append` defaults to `Unspecified`, meaning no SameSite attribute added to the cookie and the client will use its default behavior (Lax for new

browsers, None for old ones). The following code shows how to change the cookie SameSite value to `SameSiteMode.Lax`:

C#

```
HttpContext.Response.Cookies.Append(  
    "name", "value",  
    new CookieOptions() { SameSite = SameSiteMode.Lax });
```

All ASP.NET Core components that emit cookies override the preceding defaults with settings appropriate for their scenarios. The overridden preceding default values haven't changed.

 Expand table

Component	cookie	Default
CookieBuilder	SameSite	Unspecified
Session	SessionOptions.Cookie	Lax
CookieTempDataProvider	CookieTempDataProviderOptions.Cookie	Lax
IAntiforgery	AntiforgeryOptions.Cookie	Strict
Cookie Authentication	CookieAuthenticationOptions.Cookie	Lax
AddTwitter	TwitterOptions.StateCookie	Lax
RemoteAuthenticationHandler<TOptions>	RemoteAuthenticationOptions.CorrelationCookie	None
AddOpenIdConnect	OpenIdConnectOptions.NonceCookie	None
HttpContext.Response.Cookies.Append	CookieOptions	Unspecified

ASP.NET Core 3.1 and later provides the following SameSite support:

- Redefines the behavior of `SameSiteMode.None` to emit `SameSite=None`
- Adds a new value `SameSiteMode.Unspecified` to omit the SameSite attribute.
- All cookies APIs default to `Unspecified`. Some components that use cookies set values more specific to their scenarios. See the table above for examples.

In ASP.NET Core 3.0 and later the SameSite defaults were changed to avoid conflicting with inconsistent client defaults. The following APIs have changed the default from `SameSiteMode.Lax` to `-1` to avoid emitting a SameSite attribute for these cookies:

- [CookieOptions](#) used with [HttpContext.Response.Cookies.Append](#)
- [CookieBuilder](#) used as a factory for `CookieOptions`

- [CookiePolicyOptions.MinimumSameSitePolicy](#)

History and changes

SameSite support was first implemented in ASP.NET Core in 2.0 using the [2016 draft standard](#). The 2016 standard was opt-in. ASP.NET Core opted-in by setting several cookies to `Lax` by default. After encountering several [issues](#) with authentication, most SameSite usage was [disabled](#).

[Patches](#) were issued in November 2019 to update from the 2016 standard to the 2019 standard. The [2019 draft of the SameSite specification](#):

- Is **not** backwards compatible with the 2016 draft. For more information, see [Supporting older browsers](#) in this document.
- Specifies cookies are treated as `SameSite=Lax` by default.
- Specifies cookies that explicitly assert `SameSite=None` in order to enable cross-site delivery should be marked as `Secure`. `None` is a new entry to opt out.
- Is supported by patches issued for ASP.NET Core 2.1, 2.2, and 3.0. ASP.NET Core 3.1 and later has additional SameSite support.
- Is scheduled to be enabled by [Chrome](#) by default in [Feb 2020](#). Browsers started moving to this standard in 2019.

APIs impacted by the change from the 2016 SameSite draft standard to the 2019 draft standard

- [Http.SameSiteMode](#)
- [CookieOptions.SameSite](#)
- [CookieBuilder.SameSite](#)
- [CookiePolicyOptions.MinimumSameSitePolicy](#)
- [Microsoft.Net.Http.Headers.SameSiteMode](#)
- [Microsoft.Net.Http.Headers.SetCookieHeaderValue.SameSite](#)

Supporting older browsers

The 2016 SameSite standard mandated that unknown values must be treated as `SameSite=Strict` values. Apps accessed from older browsers which support the 2016 SameSite standard may break when they get a SameSite property with a value of `None`. Web apps must implement browser detection if they intend to support older browsers. ASP.NET Core doesn't implement browser detection because User-Agents values are highly

volatile and change frequently. An extension point in [Microsoft.AspNetCore.CookiePolicy](#) allows plugging in User-Agent specific logic.

In `Program.cs`, add code that calls [UseCookiePolicy](#) before calling [UseAuthentication](#) or *any* method that writes cookies:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.Configure<CookiePolicyOptions>(options =>
{
    options.MinimumSameSitePolicy = SameSiteMode.Unspecified;
    options.OnAppendCookie = cookieContext =>
        CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
    options.OnDeleteCookie = cookieContext =>
        CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
});

void CheckSameSite(HttpContext httpContext, CookieOptions options)
{
    if (options.SameSite == SameSiteMode.None)
    {
        var userAgent = httpContext.Request.Headers["User-Agent"].ToString();
        if (MyUserAgentDetectionLib.DisallowsSameSiteNone(userAgent))
        {
            options.SameSite = SameSiteMode.Unspecified;
        }
    }
}

builder.Services.AddRazorPages();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseCookiePolicy();
app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

In `Program.cs`, add code similar to the following highlighted code:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.Configure<CookiePolicyOptions>(options =>
{
    options.MinimumSameSitePolicy = SameSiteMode.Unspecified;
    options.OnAppendCookie = cookieContext =>
        CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
    options.OnDeleteCookie = cookieContext =>
        CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
});

void CheckSameSite(HttpContext httpContext, CookieOptions options)
{
    if (options.SameSite == SameSiteMode.None)
    {
        var userAgent = httpContext.Request.Headers["User-Agent"].ToString();
        if (MyUserAgentDetectionLib.DisallowsSameSiteNone(userAgent))
        {
            options.SameSite = SameSiteMode.Unspecified;
        }
    }
}

builder.Services.AddRazorPages();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseCookiePolicy();
app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

In the preceding sample, `MyUserAgentDetectionLib.DisallowsSameSiteNone` is a user supplied library that detects if the user agent doesn't support SameSite `None`:

C#

```
if (MyUserAgentDetectionLib.DisallowsSameSiteNone(userAgent))
{
    options.SameSite = SameSiteMode.Unspecified;
}
```

The following code shows a sample `DisallowsSameSiteNone` method:

Warning

The following code is for demonstration only:

- It should not be considered complete.
- It is not maintained or supported.

C#

```
public static bool DisallowsSameSiteNone(string userAgent)
{
    // Check if a null or empty string has been passed in, since this
    // will cause further interrogation of the useragent to fail.
    if (String.IsNullOrEmpty(userAgent))
        return false;

    // Cover all iOS based browsers here. This includes:
    // - Safari on iOS 12 for iPhone, iPod Touch, iPad
    // - WkWebView on iOS 12 for iPhone, iPod Touch, iPad
    // - Chrome on iOS 12 for iPhone, iPod Touch, iPad
    // All of which are broken by SameSite=None, because they use the iOS
networking
    // stack.
    if (userAgent.Contains("CPU iPhone OS 12") ||
        userAgent.Contains("iPad; CPU OS 12"))
    {
        return true;
    }

    // Cover Mac OS X based browsers that use the Mac OS networking stack.
    // This includes:
    // - Safari on Mac OS X.
    // This does not include:
    // - Chrome on Mac OS X
    // Because they do not use the Mac OS networking stack.
    if (userAgent.Contains("Macintosh; Intel Mac OS X 10_14") &&
        userAgent.Contains("Version/") && userAgent.Contains("Safari"))
    {
        return true;
    }

    // Cover Chrome 50-69, because some versions are broken by SameSite=None,
```

```
// and none in this range require it.
// Note: this covers some pre-Chromium Edge versions,
// but pre-Chromium Edge does not require SameSite=None.
if (userAgent.Contains("Chrome/5") || userAgent.Contains("Chrome/6"))
{
    return true;
}

return false;
}
```

Test apps for SameSite problems

Apps that interact with remote sites such as through third-party login need to:

- Test the interaction on multiple browsers.
- Apply the [CookiePolicy browser detection and mitigation](#) discussed in this document.

Test web apps using a client version that can opt-in to the new SameSite behavior.

Chrome, Firefox, and Chromium Edge all have new opt-in feature flags that can be used for testing. After your app applies the SameSite patches, test it with older client versions, especially Safari. For more information, see [Supporting older browsers](#) in this document.

Test with Chrome

Chrome 78+ gives misleading results because it has a temporary mitigation in place. The Chrome 78+ temporary mitigation allows cookies less than two minutes old. Chrome 76 or 77 with the appropriate test flags enabled provides more accurate results. To test the new SameSite behavior toggle `chrome://flags/#same-site-by-default-cookies` to **Enabled**. Older versions of Chrome (75 and below) are reported to fail with the new `None` setting. See [Supporting older browsers](#) in this document.

Google does not make older chrome versions available. Follow the instructions at [Download Chromium](#) to test older versions of Chrome. Do **not** download Chrome from links provided by searching for older versions of chrome.

- [Chromium 76 Win64](#)
- [Chromium 74 Win64](#)

Starting in Canary version `80.0.3975.0`, the Lax+POST temporary mitigation can be disabled for testing purposes using the new flag `--enable-features=SameSiteDefaultChecksMethodRigorously` to allow testing of sites and services in the eventual end state of the feature where the mitigation has been removed. For more information, see The Chromium Projects [SameSite Updates](#)

Test with Safari

Safari 12 strictly implemented the prior draft and fails when the new `None` value is in a cookie. `None` is avoided via the browser detection code [Supporting older browsers](#) in this document. Test Safari 12, Safari 13, and WebKit based OS style logins using MSAL, ADAL or whatever library you are using. The problem is dependent on the underlying OS version. OSX Mojave (10.14) and iOS 12 are known to have compatibility problems with the new SameSite behavior. Upgrading the OS to OSX Catalina (10.15) or iOS 13 fixes the problem. Safari does not currently have an opt-in flag for testing the new spec behavior.

Test with Firefox

Firefox support for the new standard can be tested on version 68+ by opting in on the `about:config` page with the feature flag `network.cookie.sameSite.laxByDefault`. There haven't been reports of compatibility issues with older versions of Firefox.

Test with Edge browser

Edge supports the old SameSite standard. Edge version 44 doesn't have any known compatibility problems with the new standard.

Test with Edge (Chromium)

SameSite flags are set on the `edge://flags/#same-site-by-default-cookies` page. No compatibility issues were discovered with Edge Chromium.


Test with Electron

Versions of Electron include older versions of Chromium. For example, the version of Electron used by Teams is Chromium 66, which exhibits the older behavior. You must perform your own compatibility testing with the version of Electron your product uses. See [Supporting older browsers](#) in the following section.

Additional resources

- [Chromium Blog: Developers: Get Ready for New SameSite=None; Secure Cookie Settings](#) [↗](#)
- [SameSite cookies explained](#) [↗](#)
- [November 2019 Patches](#) [↗](#)

 Expand table

Sample	Document
.NET Core Razor Pages 	ASP.NET Core 3.1 Razor Pages SameSite cookie sample

Client IP safelist for ASP.NET Core

Article • 10/25/2024

By [Damien Bowden](#) and [Tom Dykstra](#)

This article shows three ways to implement an IP address safelist (also known as an allow list) in an ASP.NET Core app. An accompanying sample app demonstrates all three approaches. You can use:

- Middleware to check the remote IP address of every request.
- MVC action filters to check the remote IP address of requests for specific controllers or action methods.
- Razor Pages filters to check the remote IP address of requests for Razor pages.

In each case, a string containing approved client IP addresses is stored in an app setting. The middleware or filter:

- Parses the string into an array.
- Checks if the remote IP address exists in the array.

Access is allowed if the array contains the IP address. Otherwise, an HTTP 403 Forbidden status code is returned.

[View or download sample code](#) (how to download)

IP address safelist

In the sample app, the IP address safelist is:

- Defined by the `AdminSafeList` property in the `appsettings.json` file.
- A semicolon-delimited string that may contain both [Internet Protocol version 4 \(IPv4\)](#) and [Internet Protocol version 6 \(IPv6\)](#) addresses.

JSON

```
{  
  "AdminSafeList": "127.0.0.1;192.168.1.5;::1",  
  "Logging": {
```

In the preceding example, the IPv4 addresses of `127.0.0.1` and `192.168.1.5` and the IPv6 loopback address of `::1` (compressed format for `0:0:0:0:0:0:0:1`) are allowed.

Middleware

The `Startup.Configure` method adds the custom `AdminSafeListMiddleware` middleware type to the app's request pipeline. The safelist is retrieved with the .NET Core configuration provider and is passed as a constructor parameter.

C#

```
app.UseMiddleware<AdminSafeListMiddleware>(Configuration["AdminSafeList"]);
```

The middleware parses the string into an array and searches for the remote IP address in the array. If the remote IP address isn't found, the middleware returns HTTP 403 Forbidden. This validation process is bypassed for HTTP GET requests.

C#

```
public class AdminSafeListMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger<AdminSafeListMiddleware> _logger;
    private readonly byte[][] _safelist;

    public AdminSafeListMiddleware(
        RequestDelegate next,
        ILogger<AdminSafeListMiddleware> logger,
        string safelist)
    {
        var ips = safelist.Split(';');
        _safelist = new byte[ips.Length][];
        for (var i = 0; i < ips.Length; i++)
        {
            _safelist[i] = IPAddress.Parse(ips[i]).GetAddressBytes();
        }

        _next = next;
        _logger = logger;
    }

    public async Task Invoke(HttpContext context)
    {
        if (context.Request.Method != HttpMethod.Get.Method)
        {
            var remoteIp = context.Connection.RemoteIpAddress;
            _logger.LogDebug("Request from Remote IP address: {RemoteIp}",
remoteIp);

            var bytes = remoteIp.GetAddressBytes();
            var badIp = true;
            foreach (var address in _safelist)
            {
```

```

        if (address.SequenceEqual(bytes))
        {
            badIp = false;
            break;
        }
    }

    if (badIp)
    {
        _logger.LogWarning(
            "Forbidden Request from Remote IP address: {RemoteIp}",
remoteIp);
        context.Response.StatusCode = (int)
HttpStatusCode.Forbidden;
        return;
    }
}

await _next.Invoke(context);
}
}

```

Action filter

If you want safelist-driven access control for specific MVC controllers or action methods, use an action filter. For example:

C#

```

public class ClientIpCheckActionFilter : ActionFilterAttribute
{
    private readonly ILogger _logger;
    private readonly string _safelist;

    public ClientIpCheckActionFilter(string safelist, ILogger logger)
    {
        _safelist = safelist;
        _logger = logger;
    }

    public override void OnActionExecuting(ActionExecutingContext context)
    {
        var remoteIp = context.HttpContext.Connection.RemoteIpAddress;
        _logger.LogDebug("Remote IpAddress: {RemoteIp}", remoteIp);
        var ip = _safelist.Split(';');
        var badIp = true;

        if (remoteIp.IsIPv4MappedToIPv6)
        {
            remoteIp = remoteIp.MapToIPv4();
        }
    }
}

```

```

        foreach (var address in ip)
        {
            var testIp = IPAddress.Parse(address);

            if (testIp.Equals(remoteIp))
            {
                badIp = false;
                break;
            }
        }

        if (badIp)
        {
            _logger.LogWarning("Forbidden Request from IP: {RemoteIp}",
remoteIp);
            context.Result = new
StatusCodeResult(StatusCode.Status403Forbidden);
            return;
        }

        base.OnActionExecuting(context);
    }
}

```

In `Startup.ConfigureServices`, add the action filter to the MVC filters collection. In the following example, a `ClientIpCheckActionFilter` action filter is added. A safelist and a console logger instance are passed as constructor parameters.

```

C#

services.AddScoped<ClientIpCheckActionFilter>(container =>
{
    var loggerFactory = container.GetRequiredService<ILoggerFactory>();
    var logger = loggerFactory.CreateLogger<ClientIpCheckActionFilter>();

    return new ClientIpCheckActionFilter(
        Configuration["AdminSafeList"], logger);
});

```

The action filter can then be applied to a controller or action method with the `[ServiceFilter]` attribute:

```

C#

[ServiceFilter(typeof(ClientIpCheckActionFilter))]
[HttpGet]
public IEnumerable<string> Get()

```

In the sample app, the action filter is applied to the controller's `Get` action method.

When you test the app by sending:

- An HTTP GET request, the `[ServiceFilter]` attribute validates the client IP address. If access is allowed to the `Get` action method, a variation of the following console output is produced by the action filter and action method:

```
dbug: ClientIpSafelistComponents.Filters.ClientIpCheckActionFilter[0]
      Remote IpAddress: ::1
dbug: ClientIpAspNetCore.Controllers.ValuesController[0]
      successful HTTP GET
```

- An HTTP request verb other than GET, the `AdminSafeListMiddleware` middleware validates the client IP address.

Razor Pages filter

If you want safelist-driven access control for a Razor Pages app, use a Razor Pages filter. For example:

C#

```
public class ClientIpCheckPageFilter : IPageFilter
{
    private readonly ILogger _logger;
    private readonly IPAddress[] _safelist;

    public ClientIpCheckPageFilter(
        string safelist,
        ILogger logger)
    {
        var ips = safelist.Split(';');
        _safelist = new IPAddress[ips.Length];
        for (var i = 0; i < ips.Length; i++)
        {
            _safelist[i] = IPAddress.Parse(ips[i]);
        }

        _logger = logger;
    }

    public void OnPageHandlerExecuting(PageHandlerExecutingContext context)
    {
        var remoteIp = context.HttpContext.Connection.RemoteIpAddress;
        if (remoteIp.IsIPv4MappedToIPv6)
        {

```

```

        remoteIp = remoteIp.MapToIPv4();
    }
    _logger.LogDebug(
        "Remote IPAddress: {RemoteIp}", remoteIp);

    var badIp = true;
    foreach (var testIp in _safelist)
    {
        if (testIp.Equals(remoteIp))
        {
            badIp = false;
            break;
        }
    }

    if (badIp)
    {
        _logger.LogWarning(
            "Forbidden Request from Remote IP address: {RemoteIp}",
remoteIp);
        context.Result = new
StatusCodeResult(StatusCode.Status403Forbidden);
        return;
    }
}

public void OnPageHandlerExecuted(PageHandlerExecutedContext context)
{
}

public void OnPageHandlerSelected(PageHandlerSelectedContext context)
{
}
}

```

In `Startup.ConfigureServices`, enable the Razor Pages filter by adding it to the MVC filters collection. In the following example, a `ClientIpCheckPageFilter` Razor Pages filter is added. A safelist and a console logger instance are passed as constructor parameters.

C#

```

services.AddRazorPages()
    .AddMvcOptions(options =>
    {
        var logger = LoggerFactory.Create(builder => builder.AddConsole())
            .CreateLogger<ClientIpCheckPageFilter>();
        var filter = new ClientIpCheckPageFilter(
            Configuration["AdminSafeList"], logger);

        options.Filters.Add(filter);
    });

```


When the sample app's *Index* Razor page is requested, the Razor Pages filter validates the client IP address. The filter produces a variation of the following console output:

```
debug: ClientIpSafelistComponents.Filters.ClientIpCheckPageFilter[0]  
      Remote IPAddress: ::1
```

Additional resources

- [ASP.NET Core Middleware](#)
- [Action filters](#)
- [Filter methods for Razor Pages in ASP.NET Core](#)

ASP.NET Core performance

Article • 09/18/2024

The following articles provide information about how to optimize the performance of ASP.NET Core apps:

- [ASP.NET Core Blazor performance best practices](#)
- [ASP.NET Core Best Practices](#)
- [Overview of caching in ASP.NET Core](#)
- [Rate limiting middleware in ASP.NET Core](#)
- [Memory management and patterns in ASP.NET Core](#)
- [Scaling ASP.NET Core Apps on Azure](#)
- [Object reuse with ObjectPool in ASP.NET Core](#)
- [Response compression in ASP.NET Core](#)
- [Performance Diagnostics Tools](#)
- [ASP.NET Core load/stress testing](#)
- [Request timeouts middleware in ASP.NET Core](#)
- [Short-circuit middleware after routing](#)

Overview of caching in ASP.NET Core

Article • 05/21/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

By [Rick Anderson](#) and [Tom Dykstra](#)

In-memory caching

In-memory caching uses server memory to store cached data. This type of caching is suitable for a single server or multiple servers using session affinity. Session affinity is also known as *sticky sessions*. Session affinity means that the requests from a client are always routed to the same server for processing.

For more information, see [Cache in-memory in ASP.NET Core](#) and [Troubleshoot Azure Application Gateway session affinity issues](#).

Distributed Cache

Use a distributed cache to store data when the app is hosted in a cloud or server farm. The cache is shared across the servers that process requests. A client can submit a request that's handled by any server in the group if cached data for the client is available. ASP.NET Core works with SQL Server, [Redis](#), and [NCache](#) distributed caches.

For more information, see [Distributed caching in ASP.NET Core](#).

HybridCache

The [HybridCache](#) API bridges some gaps in the [IDistributedCache](#) and [IMemoryCache](#) APIs. `HybridCache` is an abstract class with a default implementation that handles most aspects of saving to cache and retrieving from cache.

Features

`HybridCache` has the following features that the other APIs don't have:

- A unified API for both in-process and out-of-process caching.

`HybridCache` is designed to be a drop-in replacement for existing `IDistributedCache` and `IMemoryCache` usage, and it provides a simple API for adding new caching code. If the app has an `IDistributedCache` implementation, the `HybridCache` service uses it for secondary caching. This two-level caching strategy allows `HybridCache` to provide the speed of an in-memory cache and the durability of a distributed or persistent cache.

- Stampede protection.

Cache stampede happens when a frequently used cache entry is revoked, and too many requests try to repopulate the same cache entry at the same time.

`HybridCache` combines concurrent operations, ensuring that all requests for a given response wait for the first request to populate the cache.

- Configurable serialization.

Serialization is configured as part of registering the service, with support for type-specific and generalized serializers via the `WithSerializer` and `WithSerializerFactory` methods, chained from the `AddHybridCache` call. By default, the service handles `string` and `byte[]` internally, and uses `System.Text.Json` for everything else. It can be configured for other types of serializers, such as protobuf or XML.

To see the relative simplicity of the `HybridCache` API, compare code that uses it to code that uses `IDistributedCache`. Here's an example of what using `IDistributedCache` looks like:

C#

```
public class SomeService(IDistributedCache cache)
{
    public async Task<SomeInformation> GetSomeInformationAsync(
        (string name, int id, CancellationToken token = default)
    {
        var key = $"someinfo:{name}:{id}"; // Unique key for this
        combination.
        var bytes = await cache.GetAsync(key, token); // Try to get from
        cache.
        SomeInformation info;
        if (bytes is null)
```

```

    {
        // Cache miss; get the data from the real source.
        info = await SomeExpensiveOperationAsync(name, id, token);

        // Serialize and cache it.
        bytes = SomeSerializer.Serialize(info);
        await cache.SetAsync(key, bytes, token);
    }
    else
    {
        // Cache hit; deserialize it.
        info = SomeSerializer.Deserialize<SomeInformation>(bytes);
    }
    return info;
}

// This is the work we're trying to cache.
private async Task<SomeInformation> SomeExpensiveOperationAsync(string
name, int id,
    CancellationToken token = default)
    { /* ... */ }
}

```

That's a lot of work to get right each time, including things like serialization. And in the "cache miss" scenario, you could end up with multiple concurrent threads, all getting a cache miss, all fetching the underlying data, all serializing it, and all sending that data to the cache.

Here's equivalent code using `HybridCache`:

```

C#

public class SomeService(HybridCache cache)
{
    public async Task<SomeInformation> GetSomeInformationAsync(
        (string name, int id, CancellationToken token = default)
    {
        return await cache.GetOrCreateAsync(
            $"someinfo:{name}:{id}", // Unique key for this entry.
            async cancel => await SomeExpensiveOperationAsync(name, id,
cancel),
            token: token
        );
    }
}

```

The code is simpler and the library provides stampede protection and other features that `IDistributedCache` doesn't.

Compatibility

The `HybridCache` library supports older .NET runtimes, down to .NET Framework 4.7.2 and .NET Standard 2.0.

Additional resources

For more information, see the following resources:

- [HybridCache library in ASP.NET Core](#)
- [GitHub issue dotnet/aspnetcore #54647](#) ↗
- [HybridCache source code](#) ↗

Response caching

The Response caching middleware:

- Enables caching server responses based on [HTTP cache headers](#) ↗. Implements the standard HTTP caching semantics. Caches based on HTTP cache headers like proxies do.
- Is typically not beneficial for UI apps such as Razor Pages because browsers generally set request headers that prevent caching. [Output caching](#), which is available in ASP.NET Core 7.0 and later, benefits UI apps. With output caching, configuration decides what should be cached independently of HTTP headers.
- May be beneficial for public GET or HEAD API requests from clients where the [Conditions for caching](#) are met.

To test response caching, use [Fiddler](#) ↗, or another tool that can explicitly set request headers. Setting headers explicitly is preferred for testing caching. For more information, see [Troubleshooting](#).

For more information, see [Response caching in ASP.NET Core](#).

Output caching

The output caching middleware enables caching of HTTP responses. Output caching differs from [response caching](#) in the following ways:

- The caching behavior is configurable on the server.

Response caching behavior is defined by HTTP headers. For example, when you visit a website with Chrome or Edge, the browser automatically sends a `Cache-`

`control: max-age=0` header. This header effectively disables response caching, since the server follows the directions provided by the client. A new response is returned for every request, even if the server has a fresh cached response. With output caching the client doesn't override the caching behavior that you configure on the server.

- The cache storage medium is extensible.

Memory is used by default. Response caching is limited to memory.

- You can programmatically invalidate selected cache entries.

Response caching's dependence on HTTP headers leaves you with few options for invalidating cache entries.

- Resource locking mitigates the risk of cache stampede and thundering herd.

Cache stampede happens when a frequently used cache entry is revoked, and too many requests try to repopulate the same cache entry at the same time.

Thundering herd is similar: a burst of requests for the same response that isn't already in a cache entry. Resource locking ensures that all requests for a given response wait for the first request to populate the cache. Response caching doesn't have a resource locking feature.

- Cache revalidation minimizes bandwidth usage.

Cache revalidation means the server can return a `304 Not Modified` HTTP status code instead of a cached response body. This status code informs the client that the response to the request is unchanged from what was previously received. Response caching doesn't do cache revalidation.

For more information, see [Output caching middleware in ASP.NET Core](#).

Cache Tag Helper

Cache the content from an MVC view or Razor Page with the Cache Tag Helper. The Cache Tag Helper uses in-memory caching to store data.

For more information, see [Cache Tag Helper in ASP.NET Core MVC](#).

Distributed Cache Tag Helper

Cache the content from an MVC view or Razor Page in distributed cloud or web farm scenarios with the Distributed Cache Tag Helper. The Distributed Cache Tag Helper uses SQL Server, [Redis](#), or [NCache](#) to store data.

For more information, see [Distributed Cache Tag Helper in ASP.NET Core](#).

Cache in-memory in ASP.NET Core

Article • 04/23/2024

By [Rick Anderson](#), [John Luo](#), and [Steve Smith](#)

Caching can significantly improve the performance and scalability of an app by reducing the work required to generate content. Caching works best with data that changes infrequently **and** is expensive to generate. Caching makes a copy of data that can be returned much faster than from the source. Apps should be written and tested to **never** depend on cached data.

ASP.NET Core supports several different caches. The simplest cache is based on the [IMemoryCache](#). `IMemoryCache` represents a cache stored in the memory of the web server. Apps running on a server farm (multiple servers) should ensure sessions are sticky when using the in-memory cache. Sticky sessions ensure that requests from a client all go to the same server. For example, Azure Web apps use [Application Request Routing](#) (ARR) to route all requests to the same server.

Non-sticky sessions in a web farm require a [distributed cache](#) to avoid cache consistency problems. For some apps, a distributed cache can support higher scale-out than an in-memory cache. Using a distributed cache offloads the cache memory to an external process.

The in-memory cache can store any object. The distributed cache interface is limited to `byte[]`. The in-memory and distributed cache store cache items as key-value pairs.

System.Runtime.Caching/MemoryCache

[System.Runtime.Caching/MemoryCache](#) (NuGet package) can be used with:

- .NET Standard 2.0 or later.
- Any [.NET implementation](#) that targets .NET Standard 2.0 or later. For example, ASP.NET Core 3.1 or later.
- .NET Framework 4.5 or later.

[Microsoft.Extensions.Caching.Memory](#) / `IMemoryCache` (described in this article) is recommended over `System.Runtime.Caching/MemoryCache` because it's better integrated into ASP.NET Core. For example, `IMemoryCache` works natively with ASP.NET Core [dependency injection](#).

Use `System.Runtime.Caching/MemoryCache` as a compatibility bridge when porting code from ASP.NET 4.x to ASP.NET Core.

Cache guidelines

- Code should always have a fallback option to fetch data and **not** depend on a cached value being available.
- The cache uses a scarce resource, memory. Limit cache growth:
 - Do **not** insert external input into the cache. As an example, using arbitrary user-provided input as a cache key is not recommended since the input might consume an unpredictable amount of memory.
 - Use expirations to limit cache growth.
 - [Use `SetSize`, `Size`, and `SizeLimit` to limit cache size](#). The ASP.NET Core runtime does **not** limit cache size based on memory pressure. It's up to the developer to limit cache size.

Use `IMemoryCache`

Warning

Using a *shared* memory cache from [Dependency Injection](#) and calling `SetSize`, `Size`, or `SizeLimit` to limit cache size can cause the app to fail. When a size limit is set on a cache, all entries must specify a size when being added. This can lead to issues since developers may not have full control on what uses the shared cache. When using `SetSize`, `Size`, or `SizeLimit` to limit cache, create a cache singleton for caching. For more information and an example, see [Use `SetSize`, `Size`, and `SizeLimit` to limit cache size](#). A shared cache is one shared by other frameworks or libraries.

In-memory caching is a *service* that's referenced from an app using [Dependency Injection](#). Request the `IMemoryCache` instance in the constructor:

C#

```
public class IndexModel : PageModel
{
    private readonly IMemoryCache _memoryCache;

    public IndexModel(IMemoryCache memoryCache) =>
        _memoryCache = memoryCache;
```

```
// ...
```

The following code uses `TryGetValue` to check if a time is in the cache. If a time isn't cached, a new entry is created and added to the cache with `Set`:

C#

```
public void OnGet()
{
    CurrentDateTime = DateTime.Now;

    if (!_memoryCache.TryGetValue(CacheKeys.Entry, out DateTime cacheValue))
    {
        cacheValue = CurrentDateTime;

        var cacheEntryOptions = new MemoryCacheEntryOptions()
            .SetSlidingExpiration(TimeSpan.FromSeconds(3));

        _memoryCache.Set(CacheKeys.Entry, cacheValue, cacheEntryOptions);
    }

    CacheCurrentDateTime = cacheValue;
}
```

In the preceding code, the cache entry is configured with a sliding expiration of three seconds. If the cache entry isn't accessed for more than three seconds, it gets evicted from the cache. Each time the cache entry is accessed, it remains in the cache for a further 3 seconds. The `CacheKeys` class is part of the download sample.

The current time and the cached time are displayed:

CSHTML

```
<ul>
    <li>Current Time: @Model.CurrentDateTime</li>
    <li>Cached Time: @Model.CacheCurrentDateTime</li>
</ul>
```

The following code uses the `Set` extension method to cache data for a relative time without `MemoryCacheEntryOptions`:

C#

```
_memoryCache.Set(CacheKeys.Entry, DateTime.Now, TimeSpan.FromDays(1));
```

In the preceding code, the cache entry is configured with a relative expiration of one day. The cache entry gets evicted from the cache after one day, even if it's accessed within this timeout period.

The following code uses [GetOrCreate](#) and [GetOrCreateAsync](#) to cache data.

C#

```
public void OnGetCacheGetOrCreate()
{
    var cachedValue = _memoryCache.GetOrCreate(
        CacheKeys.Entry,
        cacheEntry =>
        {
            cacheEntry.SlidingExpiration = TimeSpan.FromSeconds(3);
            return DateTime.Now;
        });

    // ...
}

public async Task OnGetCacheGetOrCreateAsync()
{
    var cachedValue = await _memoryCache.GetOrCreateAsync(
        CacheKeys.Entry,
        cacheEntry =>
        {
            cacheEntry.SlidingExpiration = TimeSpan.FromSeconds(3);
            return Task.FromResult(DateTime.Now);
        });

    // ...
}
```

The following code calls [Get](#) to fetch the cached time:

C#

```
var cacheEntry = _memoryCache.Get<DateTime?>(CacheKeys.Entry);
```

The following code gets or creates a cached item with absolute expiration:

C#

```
var cachedValue = _memoryCache.GetOrCreate(
    CacheKeys.Entry,
    cacheEntry =>
    {
        cacheEntry.AbsoluteExpirationRelativeToNow =
            TimeSpan.FromSeconds(20);
```

```
        return DateTime.Now;
    });
```

A cached item set with only a sliding expiration is at risk of never expiring. If the cached item is repeatedly accessed within the sliding expiration interval, the item never expires. Combine a sliding expiration with an absolute expiration to guarantee the item expires. The absolute expiration sets an upper bound on how long the item can be cached while still allowing the item to expire earlier if it isn't requested within the sliding expiration interval. If either the sliding expiration interval *or* the absolute expiration time pass, the item is evicted from the cache.

The following code gets or creates a cached item with both sliding *and* absolute expiration:

C#

```
var cachedValue = _memoryCache.GetOrCreate(
    CacheKeys.CallbackEntry,
    cacheEntry =>
    {
        cacheEntry.SlidingExpiration = TimeSpan.FromSeconds(3);
        cacheEntry.AbsoluteExpirationRelativeToNow =
        TimeSpan.FromSeconds(20);
        return DateTime.Now;
    });
```

The preceding code guarantees the data won't be cached longer than the absolute time.

[GetOrCreate](#), [GetOrCreateAsync](#), and [Get](#) are extension methods in the [CacheExtensions](#) class. These methods extend the capability of [IMemoryCache](#).

MemoryCacheEntryOptions

The following example:

- Sets the cache priority to [CacheItemPriority.NeverRemove](#).
- Sets a [PostEvictionDelegate](#) that gets called after the entry is evicted from the cache. The callback is run on a different thread from the code that removes the item from the cache.

C#

```
public void OnGetCacheRegisterPostEvictionCallback()
{
    var memoryCacheEntryOptions = new MemoryCacheEntryOptions()
```

```

        .SetPriority(CacheItemPriority.NeverRemove)
        .RegisterPostEvictionCallback(PostEvictionCallback, _memoryCache);

        _memoryCache.Set(CacheKeys.CallbackEntry, DateTime.Now,
memoryCacheEntryOptions);
    }

    private static void PostEvictionCallback(
        object cacheKey, object cacheValue, EvictionReason evictionReason,
        object state)
    {
        var memoryCache = (IMemoryCache)state;

        memoryCache.Set(
            CacheKeys.CallbackMessage,
            $"Entry {cacheKey} was evicted: {evictionReason}.");
    }

```

Use SetSize, Size, and SizeLimit to limit cache size

A `MemoryCache` instance may optionally specify and enforce a size limit. The cache size limit doesn't have a defined unit of measure because the cache has no mechanism to measure the size of entries. If the cache size limit is set, all entries must specify size. The ASP.NET Core runtime doesn't limit cache size based on memory pressure. It's up to the developer to limit cache size. The size specified is in units the developer chooses.

For example:

- If the web app was primarily caching strings, each cache entry size could be the string length.
- The app could specify the size of all entries as 1, and the size limit is the count of entries.

If `SizeLimit` isn't set, the cache grows without bound. The ASP.NET Core runtime doesn't trim the cache when system memory is low. Apps must be architected to:

- Limit cache growth.
- Call `Compact` or `Remove` when available memory is limited.

The following code creates a unitless fixed size `MemoryCache` accessible by [dependency injection](#):

C#

```
public class MyMemoryCache
{
    public MemoryCache Cache { get; } = new MemoryCache(
        new MemoryCacheOptions
        {
            SizeLimit = 1024
        });
}
```

`SizeLimit` doesn't have units. Cached entries must specify size in whatever units they consider most appropriate if the cache size limit has been set. All users of a cache instance should use the same unit system. An entry won't be cached if the sum of the cached entry sizes exceeds the value specified by `SizeLimit`. If no cache size limit is set, the cache size set on the entry is ignored.

The following code registers `MyMemoryCache` with the [dependency injection](#) container:

```
C#

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddSingleton<MyMemoryCache>();
```

`MyMemoryCache` is created as an independent memory cache for components that are aware of this size limited cache and know how to set cache entry size appropriately.

The size of the cache entry can be set using the [SetSize](#) extension method or the [Size](#) property:

```
C#

if (!_myMemoryCache.Cache.TryGetValue(CacheKeys.Entry, out DateTime
cacheValue))
{
    var cacheEntryOptions = new MemoryCacheEntryOptions()
        .SetSize(1);

    // cacheEntryOptions.Size = 1;

    _myMemoryCache.Cache.Set(CacheKeys.Entry, cacheValue,
cacheEntryOptions);
}
```

In the preceding code, the two highlighted lines achieve the same result of setting the size of the cache entry. `SetSize` is provided for convenience when chaining calls onto

```
new MemoryCacheOptions().
```

MemoryCache.Compact

`MemoryCache.Compact` attempts to remove the specified percentage of the cache in the following order:

- All expired items.
- Items by priority. Lowest priority items are removed first.
- Least recently used objects.
- Items with the earliest absolute expiration.
- Items with the earliest sliding expiration.

Pinned items with priority `NeverRemove` are [never removed](#). The following code removes a cache item and calls `Compact` to remove 25% of cached entries:

C#

```
_myMemoryCache.Cache.Remove(CacheKeys.Entry);  
_myMemoryCache.Cache.Compact(.25);
```

For more information, see the [Compact source on GitHub](#).

Cache dependencies

The following sample shows how to expire a cache entry if a dependent entry expires. A `CancellationToken` is added to the cached item. When `Cancel` is called on the `CancellationTokenSource`, both cache entries are evicted:

C#

```
public void OnGetCacheCreateDependent()  
{  
    var cancellationTokenSource = new CancellationTokenSource();  
  
    _memoryCache.Set(  
        CacheKeys.DependentCancellationTokenSource,  
        cancellationTokenSource);  
  
    using var parentCacheEntry = _memoryCache.CreateEntry(CacheKeys.Parent);  
  
    parentCacheEntry.Value = DateTime.Now;  
  
    _memoryCache.Set(  
        CacheKeys.Child,
```



```

        DateTime.Now,
        new CancellationChangeToken(cancellationTokenSource.Token));
    }

    public void OnGetCacheRemoveDependent()
    {
        var cancellationTokenSource = _memoryCache.Get<CancellationTokenSource>(
            CacheKeys.DependentCancellationTokenSource);

        cancellationTokenSource.Cancel();
    }

```

Using a [CancellationTokenSource](#) allows multiple cache entries to be evicted as a group. With the `using` pattern in the code above, cache entries created inside the `using` scope inherit triggers and expiration settings.

Additional notes

- Expiration doesn't happen in the background. There's no timer that actively scans the cache for expired items. Any activity on the cache (`Get`, `Set`, `Remove`) can trigger a background scan for expired items. A timer on the `CancellationTokenSource` ([CancelAfter](#)) also removes the entry and triggers a scan for expired items. The following example uses [CancellationTokenSource\(TimeSpan\)](#) for the registered token. When this token fires, it removes the entry immediately and fires the eviction callbacks:

```

C#

if (!_memoryCache.TryGetValue(CacheKeys.Entry, out DateTime
    cacheValue))
{
    cacheValue = DateTime.Now;

    var cancellationTokenSource = new CancellationTokenSource(
        TimeSpan.FromSeconds(10));

    var cacheEntryOptions = new MemoryCacheEntryOptions()
        .AddExpirationToken(
            new CancellationChangeToken(cancellationTokenSource.Token))
        .RegisterPostEvictionCallback((key, value, reason, state) =>
        {
            ((CancellationTokenSource)state).Dispose();
        }, cancellationTokenSource);

    _memoryCache.Set(CacheKeys.Entry, cacheValue, cacheEntryOptions);
}

```

- When using a callback to repopulate a cache item:
 - Multiple requests can find the cached key value empty because the callback hasn't completed.
 - This can result in several threads repopulating the cached item.
- When one cache entry is used to create another, the child copies the parent entry's expiration tokens and time-based expiration settings. The child isn't expired by manual removal or updating of the parent entry.
- Use [PostEvictionCallbacks](#) to set the callbacks that will be fired after the cache entry is evicted from the cache.
- For most apps, `IMemoryCache` is enabled. For example, calling `AddMvc`, `AddControllersWithViews`, `AddRazorPages`, `AddMvcCore().AddRazorViewEngine`, and many other `Add{Service}` methods in `Program.cs`, enables `IMemoryCache`. For apps that don't call one of the preceding `Add{Service}` methods, it may be necessary to call [AddMemoryCache](#) in `Program.cs`.

Background cache update

Use a [background service](#) such as `IHostedService` to update the cache. The background service can recompute the entries and then assign them to the cache only when they're ready.

Additional resources

- [View or download sample code](#) [↗] (how to download)
- [Distributed caching in ASP.NET Core](#)
- [Detect changes with change tokens in ASP.NET Core](#)
- [Response caching in ASP.NET Core](#)
- [Response Caching Middleware in ASP.NET Core](#)
- [Cache Tag Helper in ASP.NET Core MVC](#)
- [Distributed Cache Tag Helper in ASP.NET Core](#)

 **Note:** The author created this article with assistance from AI. [Learn more](#)

Distributed caching in ASP.NET Core

Article • 10/04/2024

By [Mohsin Nasir](#) and [smandia](#)

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

A distributed cache is a cache shared by multiple app servers, typically maintained as an external service to the app servers that access it. A distributed cache can improve the performance and scalability of an ASP.NET Core app, especially when the app is hosted by a cloud service or a server farm.

A distributed cache has several advantages over other caching scenarios where cached data is stored on individual app servers.

When cached data is distributed, the data:

- Is *coherent* (consistent) across requests to multiple servers.
- Survives server restarts and app deployments.
- Doesn't use local memory.

Distributed cache configuration is implementation specific. This article describes how to configure SQL Server and Redis distributed caches. Third party implementations are also available, such as [NCache](#) ([NCache on GitHub](#)). Regardless of which implementation is selected, the app interacts with the cache using the [IDistributedCache](#) interface.

[View or download sample code](#) ([how to download](#))

Warning

This article uses a local database that doesn't require the user to be authenticated. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production apps, see [Secure authentication flows](#).

Prerequisites

Add a package reference for the distributed cache provider used:

- For a Redis distributed cache, [Microsoft.Extensions.Caching.StackExchangeRedis](#) [↗].
- For SQL Server, [Microsoft.Extensions.Caching.SqlServer](#) [↗].
- For the NCache distributed cache, [NCache.Microsoft.Extensions.Caching.OpenSource](#) [↗].

IDistributedCache interface

The [IDistributedCache](#) interface provides the following methods to manipulate items in the distributed cache implementation:

- [Get](#), [GetAsync](#): Accepts a string key and retrieves a cached item as a `byte[]` array if found in the cache.
- [Set](#), [SetAsync](#): Adds an item (as `byte[]` array) to the cache using a string key.
- [Refresh](#), [RefreshAsync](#): Refreshes an item in the cache based on its key, resetting its sliding expiration timeout (if any).
- [Remove](#), [RemoveAsync](#): Removes a cache item based on its string key.

Establish distributed caching services

Register an implementation of [IDistributedCache](#) in `Program.cs`. Framework-provided implementations described in this topic include:

- [Distributed Redis cache](#)
- [Distributed Memory Cache](#)
- [Distributed SQL Server cache](#)
- [Distributed NCache cache](#)
- [Distributed Azure CosmosDB cache](#)

Distributed Redis Cache

We recommend production apps use the Distributed Redis Cache because it's the most performant. For more information see [Recommendations](#).

[Redis](#) [↗] is an open source in-memory data store, which is often used as a distributed cache. You can configure an [Azure Cache for Redis](#) for an Azure-hosted ASP.NET Core app, and use an Azure Cache for Redis for local development.

An app configures the cache implementation using a [RedisCache](#) instance, by calling [AddStackExchangeRedisCache](#). For [output caching](#), use

`AddStackExchangeRedisOutputCache`.

1. Create an Azure Cache for Redis.
2. Copy the Primary connection string (StackExchange.Redis) to [Configuration](#).
 - Local development: Save the connection string with [Secret Manager](#).
 - Azure: Save the connection string in a secure store such as [Azure Key Vault](#)

The following code enables the Azure Cache for Redis:

```
C#

builder.Services.AddStackExchangeRedisCache(options =>
{
    options.Configuration =
builder.Configuration.GetConnectionString("MyRedisConStr");
    options.InstanceName = "SampleInstance";
});
```

The preceding code assumes the Primary connection string (StackExchange.Redis) was saved in configuration with the key name `MyRedisConStr`.

For more information, see [Azure Cache for Redis](#).

See [this GitHub issue](#) [↗](#) for a discussion on alternative approaches to a local Redis cache.

Distributed Memory Cache

The Distributed Memory Cache ([AddDistributedMemoryCache](#)) is a framework-provided implementation of [IDistributedCache](#) that stores items in memory. The Distributed Memory Cache isn't an actual distributed cache. Cached items are stored by the app instance on the server where the app is running.

The Distributed Memory Cache is a useful implementation:

- In development and testing scenarios.
- When a single server is used in production and memory consumption isn't an issue. Implementing the Distributed Memory Cache abstracts cached data storage. It allows for implementing a true distributed caching solution in the future if multiple nodes or fault tolerance become necessary.

The sample app makes use of the Distributed Memory Cache when the app is run in the Development environment in `Program.cs`:

```
C#
```

```
builder.Services.AddDistributedMemoryCache();
```

Distributed SQL Server Cache

The Distributed SQL Server Cache implementation ([AddDistributedSqlServerCache](#)) allows the distributed cache to use a SQL Server database as its backing store. To create a SQL Server cached item table in a SQL Server instance, you can use the `sql-cache` tool. The tool creates a table with the name and schema that you specify.

Create a table in SQL Server by running the `sql-cache create` command. Provide the SQL Server instance (`Data Source`), database (`Initial Catalog`), schema (for example, `dbo`), and table name (for example, `TestCache`):

```
.NET CLI
```


```
dotnet sql-cache create "Data Source=(localdb)/MSSQLLocalDB;Initial Catalog=DistCache;Integrated Security=True;" dbo TestCache
```


A message is logged to indicate that the tool was successful:

```
Console
```

```
Table and index were created successfully.
```

The table created by the `sql-cache` tool has the following schema:

	Name	Data Type	Allow Nulls
	Id	nvarchar(449)	<input type="checkbox"/>
	Value	varbinary(MAX)	<input type="checkbox"/>
	ExpiresAtTime	datetimeoffset(7)	<input type="checkbox"/>
	SlidingExpirationInSeconds	bigint	<input checked="" type="checkbox"/>
	AbsoluteExpiration	datetimeoffset(7)	<input checked="" type="checkbox"/>

 Note

An app should manipulate cache values using an instance of [IDistributedCache](#), not a [SqlServerCache](#).

The sample app implements [SqlServerCache](#) in a non-Development environment in `Program.cs`:

C#

```
builder.Services.AddDistributedSqlServerCache(options =>
{
    options.ConnectionString = builder.Configuration.GetConnectionString(
        "DistCache_ConnectionString");
    options.SchemaName = "dbo";
    options.TableName = "TestCache";
});
```

ⓘ Note

A [ConnectionString](#) (and optionally, [SchemaName](#) and [TableName](#)) are typically stored outside of source control (for example, stored by the [Secret Manager](#) or in `appsettings.json/appsettings.{Environment}.json` files). The connection string may contain credentials that should be kept out of source control systems.

Distributed NCache Cache

[NCache](#) is an open source in-memory distributed cache developed natively in .NET and .NET Core. NCache works both locally and configured as a distributed cache cluster for an ASP.NET Core app running in Azure or on other hosting platforms.

To install and configure NCache on your local machine, see [Getting Started Guide for Windows \(.NET and .NET Core\)](#).

To configure NCache:

1. Install [NCache open source NuGet](#).
2. Configure the cache cluster in [client.nconf](#).
3. Add the following code to `Program.cs`:

C#

```
builder.Services.AddNCacheDistributedCache(configuration =>
{
    configuration.CacheName = "democache";
    configuration.EnableLogs = true;
```

```
configuration.ExceptionsEnabled = true;
});
```

Distributed Azure CosmosDB Cache

[Azure Cosmos DB](#) can be used in ASP.NET Core as a session state provider by using the `IDistributedCache` interface. Azure Cosmos DB is a fully managed NoSQL and relational database for modern app development that offers high availability, scalability, and low-latency access to data for mission-critical applications.

After installing the [Microsoft.Extensions.Caching.Cosmos](#) NuGet package, configure an Azure Cosmos DB distributed cache as follows:

Reuse an existing client

The easiest way to configure distributed cache is by reusing an existing Azure Cosmos DB client. In this case, the `CosmosClient` instance won't be disposed when the provider is disposed.

```
C#

services.AddCosmosCache((CosmosCacheOptions cacheOptions) =>
{
    cacheOptions.ContainerName = Configuration["CosmosCacheContainer"];
    cacheOptions.DatabaseName = Configuration["CosmosCacheDatabase"];
    cacheOptions.CosmosClient = existingCosmosClient;
    cacheOptions.CreateIfNotExists = true;
});
```

Create a new client

Alternatively, instantiate a new client. In this case, the `CosmosClient` instance will get disposed when the provider is disposed.

```
C#

services.AddCosmosCache((CosmosCacheOptions cacheOptions) =>
{
    cacheOptions.ContainerName = Configuration["CosmosCacheContainer"];
    cacheOptions.DatabaseName = Configuration["CosmosCacheDatabase"];
    cacheOptions.ClientBuilder = new
CosmosClientBuilder(Configuration["CosmosConnectionString"]);
    cacheOptions.CreateIfNotExists = true;
});
```


Use the distributed cache

To use the [IDistributedCache](#) interface, request an instance of [IDistributedCache](#) in the app. The instance is provided by [dependency injection \(DI\)](#).

When the sample app starts, [IDistributedCache](#) is injected into `Program.cs`. The current time is cached using [IHostApplicationLifetime](#) (for more information, see [Generic Host: IHostApplicationLifetime](#)):

C#

```
app.Lifetime.ApplicationStarted.Register(() =>
{
    var currentTimeUTC = DateTime.UtcNow.ToString();
    byte[] encodedCurrentTimeUTC =
System.Text.Encoding.UTF8.GetBytes(currentTimeUTC);
    var options = new DistributedCacheEntryOptions()
        .SetSlidingExpiration(TimeSpan.FromSeconds(20));
    app.Services.GetService<IDistributedCache>()
        .Set("cachedTimeUTC", encodedCurrentTimeUTC,
options);
});
```

The sample app injects [IDistributedCache](#) into the `IndexModel` for use by the Index page.

Each time the Index page is loaded, the cache is checked for the cached time in `OnGetAsync`. If the cached time hasn't expired, the time is displayed. If 20 seconds have elapsed since the last time the cached time was accessed (the last time this page was loaded), the page displays *Cached Time Expired*.

Immediately update the cached time to the current time by selecting the **Reset Cached Time** button. The button triggers the `OnPostResetCachedTime` handler method.

C#

```
public class IndexModel : PageModel
{
    private readonly IDistributedCache _cache;

    public IndexModel(IDistributedCache cache)
    {
        _cache = cache;
    }

    public string? CachedTimeUTC { get; set; }
    public string? ASP_Environment { get; set; }

    public async Task OnGetAsync()
```

```

{
    CachedTimeUTC = "Cached Time Expired";
    var encodedCachedTimeUTC = await _cache.GetAsync("cachedTimeUTC");

    if (encodedCachedTimeUTC != null)
    {
        CachedTimeUTC = Encoding.UTF8.GetString(encodedCachedTimeUTC);
    }

    ASP_Environment =
Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT");
    if (String.IsNullOrEmpty(ASP_Environment))
    {
        ASP_Environment = "Null, so Production";
    }
}

public async Task<IActionResult> OnPostResetCachedTime()
{
    var currentTimeUTC = DateTime.UtcNow.ToString();
    byte[] encodedCurrentTimeUTC =
Encoding.UTF8.GetBytes(currentTimeUTC);
    var options = new DistributedCacheEntryOptions()
        .SetSlidingExpiration(TimeSpan.FromSeconds(20));
    await _cache.SetAsync("cachedTimeUTC", encodedCurrentTimeUTC,
options);

    return RedirectToPage();
}
}

```

There's ***no*** need to use a Singleton or Scoped lifetime for [IDistributedCache](#) instances with the built-in implementations.

You can also create an [IDistributedCache](#) instance wherever you might need one instead of using DI, but creating an instance in code can make your code harder to test and violates the [Explicit Dependencies Principle](#).

Recommendations

When deciding which implementation of [IDistributedCache](#) is best for your app, consider the following:

- Existing infrastructure
- Performance requirements
- Cost
- Team experience

Caching solutions usually rely on in-memory storage to provide fast retrieval of cached data, but memory is a limited resource and costly to expand. Only store commonly used data in a cache.

For most apps, a Redis cache provides higher throughput and lower latency than a SQL Server cache. However, benchmarking is recommended to determine the performance characteristics of caching strategies.

When SQL Server is used as a distributed cache backing store, use of the same database for the cache and the app's ordinary data storage and retrieval can negatively impact the performance of both. We recommend using a dedicated SQL Server instance for the distributed cache backing store.

Additional resources

- [Redis Cache on Azure](#)
- [SQL Database on Azure](#)
- [ASP.NET Core IDistributedCache Provider for NCache in Web Farms [↗]](#) (NCache on [GitHub [↗]](#))
- [Repository README file for Microsoft.Extensions.Caching.Cosmos [↗]](#)
- [Cache in-memory in ASP.NET Core](#)
- [Detect changes with change tokens in ASP.NET Core](#)
- [Response caching in ASP.NET Core](#)
- [Response Caching Middleware in ASP.NET Core](#)
- [Cache Tag Helper in ASP.NET Core MVC](#)
- [Distributed Cache Tag Helper in ASP.NET Core](#)
- [Host ASP.NET Core in a web farm](#)

HybridCache library in ASP.NET Core

Article • 12/02/2024

❗ Important

`HybridCache` is currently still in preview but will be fully released *after* .NET 9.0 in a future minor release of .NET Extensions.

This article explains how to configure and use the `HybridCache` library in an ASP.NET Core app. For an introduction to the library, see [the HybridCache section of the Caching overview](#).

Get the library

Install the `Microsoft.Extensions.Caching.Hybrid` package.

.NET CLI

```
dotnet add package Microsoft.Extensions.Caching.Hybrid --version "9.0.0-preview.7.24406.2"
```

Register the service

Add the `HybridCache` service to the [dependency injection \(DI\)](#) container by calling [AddHybridCache](#) [↗](#):

C#

```
// Add services to the container.
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddAuthorization();

builder.Services.AddHybridCache();
```

The preceding code registers the `HybridCache` service with default options. The registration API can also configure [options](#) and [serialization](#).

Get and store cache entries

The `HybridCache` service provides a `GetOrCreateAsync` [↗](#) method with two overloads, taking a key and:

- A factory method.
- State, and a factory method.

The method uses the key to try to retrieve the object from the primary cache. If the item isn't found in the primary cache (a cache miss), it then checks the secondary cache if one is configured. If it doesn't find the data there (another cache miss), it calls the factory method to get the object from the data source. It then stores the object in both primary and secondary caches. The factory method is never called if the object is found in the primary or secondary cache (a cache hit).

The `HybridCache` service ensures that only one concurrent caller for a given key calls the factory method, and all other callers wait for the result of that call. The `CancellationToken` passed to `GetOrCreateAsync` represents the combined cancellation of all concurrent callers.

The main `GetOrCreateAsync` overload

The stateless overload of `GetOrCreateAsync` is recommended for most scenarios. The code to call it is relatively simple. Here's an example:

C#

```
public class SomeService(HybridCache cache)
{
    private HybridCache _cache = cache;

    public async Task<string> GetSomeInfoAsync(string name, int id,
        CancellationToken token = default)
    {
        return await _cache.GetOrCreateAsync(
            $"{name}-{id}", // Unique key to the cache entry
            async cancel => await GetDataFromTheSourceAsync(name, id,
                cancel),
            cancellationToken: token
        );
    }

    public async Task<string> GetDataFromTheSourceAsync(string name, int id,
        CancellationToken token)
    {
        string someInfo = $"someinfo-{name}-{id}";
        return someInfo;
    }
}
```

```
}  
}
```

The alternative `GetOrCreateAsync` overload

The alternative overload might reduce some overhead from [captured variables](#) and per-instance callbacks, but at the expense of more complex code. For most scenarios the performance increase doesn't outweigh the code complexity. Here's an example that uses the alternative overload:

C#

```
public class SomeService(HybridCache cache)
{
    private HybridCache _cache = cache;

    public async Task<string> GetSomeInfoAsync(string name, int id,
        CancellationToken token = default)
    {
        return await _cache.GetOrCreateAsync(
            $"{name}-{id}", // Unique key to the cache entry
            (name, id, obj: this),
            static async (state, token) =>
                await state.obj.GetDataFromTheSourceAsync(state.name, state.id,
                    token),
            cancellationTokens: token
        );
    }

    public async Task<string> GetDataFromTheSourceAsync(string name, int id,
        CancellationToken token)
    {
        string someInfo = $"someinfo-{name}-{id}";
        return someInfo;
    }
}
```

The `SetAsync` method

In many scenarios, `GetOrCreateAsync` is the only API needed. But `HybridCache` also has [SetAsync](#) to store an object in cache without trying to retrieve it first.

Remove cache entries by key

When the underlying data for a cache entry changes before it expires, remove the entry explicitly by calling [RemoveAsync](#) with the key to the entry. An [overload](#) lets you

specify a collection of key values.

When an entry is removed, it is removed from both the primary and secondary caches.

Remove cache entries by tag

Important

This feature is still under development. If you try to remove entries by tag, you will notice that it doesn't have any effect.

Tags can be used to group cache entries and invalidate them together.

Set tags when calling `GetOrCreateAsync`, as shown in the following example:

C#

```
public class SomeService(HybridCache cache)
{
    private HybridCache _cache = cache;

    public async Task<string> GetSomeInfoAsync(string name, int id,
        CancellationToken token = default)
    {
        var tags = new List<string> { "tag1", "tag2", "tag3" };
        var entryOptions = new HybridCacheEntryOptions
        {
            Expiration = TimeSpan.FromMinutes(1),
            LocalCacheExpiration = TimeSpan.FromMinutes(1)
        };
        return await _cache.GetOrCreateAsync(
            $"{name}-{id}", // Unique key to the cache entry
            async cancel => await GetDataFromTheSourceAsync(name, id,
                cancel),
            entryOptions,
            tags,
            cancellation token: token
        );
    }

    public async Task<string> GetDataFromTheSourceAsync(string name, int id,
        CancellationToken token)
    {
        string someInfo = $"someinfo-{name}-{id}";
        return someInfo;
    }
}
```

Remove all entries for a specified tag by calling [RemoveByTagAsync](#) with the tag value. An [overload](#) lets you specify a collection of tag values.

When an entry is removed, it is removed from both the primary and secondary caches.

Options

The `AddHybridCache` method can be used to configure global defaults. The following example shows how to configure some of the available options:

```
C#

// Add services to the container.
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorization();

builder.Services.AddHybridCache(options =>
{
    options.MaximumPayloadBytes = 1024 * 1024;
    options.MaximumKeyLength = 1024;
    options.DefaultEntryOptions = new HybridCacheEntryOptions
    {
        Expiration = TimeSpan.FromMinutes(5),
        LocalCacheExpiration = TimeSpan.FromMinutes(5)
    };
});
```

The `GetOrCreateAsync` method can also take a `HybridCacheEntryOptions` object to override the global defaults for a specific cache entry. Here's an example:

```
C#

public class SomeService(HybridCache cache)
{
    private HybridCache _cache = cache;

    public async Task<string> GetSomeInfoAsync(string name, int id,
        CancellationToken token = default)
    {
        var tags = new List<string> { "tag1", "tag2", "tag3" };
        var entryOptions = new HybridCacheEntryOptions
        {
            Expiration = TimeSpan.FromMinutes(1),
            LocalCacheExpiration = TimeSpan.FromMinutes(1)
        };
        return await _cache.GetOrCreateAsync(
            $"{name}-{id}", // Unique key to the cache entry
            async cancel => await GetDataFromTheSourceAsync(name, id,
```



```

cancel),
    entryOptions,
    tags,
    cancellationToken: token
);
}

public async Task<string> GetDataFromTheSourceAsync(string name, int id,
CancellationToken token)
{
    string someInfo = $"someinfo-{name}-{id}";
    return someInfo;
}
}

```

For more information about the options, see the source code:

- [HybridCacheOptions](#) [↗] class.
- [HybridCacheEntryOptions](#) [↗] class.

Limits

The following properties of `HybridCacheOptions` let you configure limits that apply to all cache entries:

- `MaximumPayloadBytes` - Maximum size of a cache entry. Default value is 1 MB. Attempts to store values over this size are logged, and the value isn't stored in cache.
- `MaximumKeyLength` - Maximum length of a cache key. Default value is 1024 characters. Attempts to store values over this size are logged, and the value isn't stored in cache.

Serialization

Use of a secondary, out-of-process cache requires serialization. Serialization is configured as part of registering the `HybridCache` service. Type-specific and general-purpose serializers can be configured via the [AddSerializer](#) [↗] and [AddSerializerFactory](#) [↗] methods, chained from the `AddHybridCache` call. By default, the library handles `string` and `byte[]` internally, and uses `System.Text.Json` for everything else. `HybridCache` can also use other serializers, such as protobuf or XML.

The following example configures the service to use a type-specific protobuf serializer:

C#

```
// Add services to the container.

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorization();

builder.Services.AddHybridCache(options =>
{
    options.DefaultEntryOptions = new HybridCacheEntryOptions
    {
        Expiration = TimeSpan.FromSeconds(10),
        LocalCacheExpiration = TimeSpan.FromSeconds(5)
    };
}).AddSerializer<SomeProtobufMessage,
    GoogleProtobufSerializer<SomeProtobufMessage>>());
```

The following example configures the service to use a general-purpose protobuf serializer that can handle many protobuf types:

```
C#

// Add services to the container.

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorization();

builder.Services.AddHybridCache(options =>
{
    options.DefaultEntryOptions = new HybridCacheEntryOptions
    {
        Expiration = TimeSpan.FromSeconds(10),
        LocalCacheExpiration = TimeSpan.FromSeconds(5)
    };
}).AddSerializerFactory<GoogleProtobufSerializerFactory>());
```

The secondary cache requires a data store, such as Redis or SqlServer. To use [Azure Cache for Redis](#), for example:

- Install the `Microsoft.Extensions.Caching.StackExchangeRedis` package.
- Create an instance of Azure Cache for Redis.
- Get a connection string that connects to the Redis instance. Find the connection string by selecting **Show access keys** on the **Overview** page in the Azure portal.
- Store the connection string in the app's configuration. For example, use a [user secrets file](#) that looks like the following JSON, with the connection string in the

`ConnectionStrings` section. Replace `<the connection string>` with the actual connection string:

JSON

```
{
  "ConnectionStrings": {
    "RedisConnectionString": "<the connection string>"
  }
}
```

- Register in DI the `IDistributedCache` implementation that the Redis package provides. To do that, call `AddStackExchangeRedisCache`, and pass in the connection string. For example:

C#

```
builder.Services.AddStackExchangeRedisCache(options =>
{
    options.Configuration =

builder.Configuration.GetConnectionString("RedisConnectionString");
});
```

- The Redis `IDistributedCache` implementation is now available from the app's DI container. `HybridCache` uses it as the secondary cache and uses the serializer configured for it.

For more information, see the [HybridCache serialization sample app](#).

Cache storage

By default `HybridCache` uses `MemoryCache` for its primary cache storage. Cache entries are stored in-process, so each server has a separate cache that is lost whenever the server process is restarted. For secondary out-of-process storage, such as Redis or SQL Server, `HybridCache` uses [the configured `IDistributedCache` implementation](#), if any. But even without an `IDistributedCache` implementation, the `HybridCache` service still provides in-process caching and [stampede protection](#).

ⓘ Note

When invalidating cache entries by key or by tags, they are invalidated in the current server and in the secondary out-of-process storage. However, the in-

memory cache in other servers isn't affected.

Optimize performance

To optimize performance, configure `HybridCache` to reuse objects and avoid `byte[]` allocations.

Reuse objects

By reusing instances, `HybridCache` can reduce the overhead of CPU and object allocations associated with per-call deserialization. This can lead to performance improvements in scenarios where the cached objects are large or accessed frequently.

In typical existing code that uses `IDistributedCache`, every retrieval of an object from the cache results in deserialization. This behavior means that each concurrent caller gets a separate instance of the object, which can't interact with other instances. The result is thread safety, as there's no risk of concurrent modifications to the same object instance.

Because much `HybridCache` usage will be adapted from existing `IDistributedCache` code, `HybridCache` preserves this behavior by default to avoid introducing concurrency bugs. However, objects are inherently thread-safe if:

- They are immutable types.
- The code doesn't modify them.

In such cases, inform `HybridCache` that it's safe to reuse instances by:

- Marking the type as `sealed`. The `sealed` keyword in C# means that the class can't be inherited.
- Applying the `[ImmutableObject(true)]` attribute to the type. The `[ImmutableObject(true)]` attribute indicates that the object's state can't be changed after it's created.

Avoid `byte[]` allocations

`HybridCache` also provides optional APIs for `IDistributedCache` implementations, to avoid `byte[]` allocations. This feature is implemented by the preview versions of the `Microsoft.Extensions.Caching.StackExchangeRedis` and `Microsoft.Extensions.Caching.SqlServer` packages. For more information, see [IBufferDistributedCache](#) [↗] Here are the .NET CLI commands to install the packages:

.NET CLI

```
dotnet add package Microsoft.Extensions.Caching.StackExchangeRedis --prerelease
```

.NET CLI

```
dotnet add package Microsoft.Extensions.Caching.SqlServer --prerelease
```

Custom HybridCache implementations

A concrete implementation of the `HybridCache` abstract class is included in the shared framework and is provided via dependency injection. But developers are welcome to provide custom implementations of the API.

Compatibility

The `HybridCache` library supports older .NET runtimes, down to .NET Framework 4.7.2 and .NET Standard 2.0.

Additional resources

For more information about `HybridCache`, see the following resources:

- GitHub issue [dotnet/aspnetcore #54647](#) [↗](#).
- [HybridCache source code](#) [↗](#)

Output caching middleware in ASP.NET Core

Article • 07/11/2024

By [Tom Dykstra](#) 



Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

This article explains how to configure output caching middleware in an ASP.NET Core app. For an introduction to output caching, see [Output caching](#).

The output caching middleware can be used in all types of ASP.NET Core apps: Minimal API, Web API with controllers, MVC, and Razor Pages. Code examples are provided for minimal APIs and controller-based APIs. The controller-based API examples show how to use attributes to configure caching. These attributes can also be used in MVC and Razor Pages apps.

The code examples refer to a [Gravatar class](#)  that generates an image and provides a "generated at" date and time. The class is defined and used only in [the sample app](#) . Its purpose is to make it easy to see when cached output is being used. For more information, see [How to download a sample](#) and [Preprocessor directives in sample code](#).

Add the middleware to the app

Add the output caching middleware to the service collection by calling [AddOutputCache](#).

Add the middleware to the request processing pipeline by calling [UseOutputCache](#).

For example:

```
C#
```

```
builder.Services.AddOutputCache();
```

C#

```
var app = builder.Build();

// Configure the HTTP request pipeline.
app.UseHttpsRedirection();
app.UseOutputCache();
app.UseAuthorization();
```

Calling `AddOutputCache` and `UseOutputCache` doesn't start caching behavior, it makes caching available. To make the app cache responses, caching must be configured as shown in the following sections.

ⓘ Note

- In apps that use CORS middleware, `UseOutputCache` must be called after `UseCors`.
- In Razor Pages apps and apps with controllers, `UseOutputCache` must be called after `UseRouting`.

Configure one endpoint or page

For minimal API apps, configure an endpoint to do caching by calling `CacheOutput`, or by applying the `[OutputCache]` attribute, as shown in the following examples:

C#

```
app.MapGet("/cached", Gravatar.WriteGravatar).CacheOutput();
app.MapGet("/attribute", [OutputCache] (context) =>
    Gravatar.WriteGravatar(context));
```

For apps with controllers, apply the `[OutputCache]` attribute to the action method as shown here:

C#

```
[ApiController]
[Route("/[controller]")]
[OutputCache]
public class CachedController : ControllerBase
{
    public async Task GetAsync()
```

```
{  
    await Gravatar.WriteGravatar(HttpContext);  
}  
}
```

For Razor Pages apps, apply the attribute to the Razor page class.

Configure multiple endpoints or pages

Create *policies* when calling `AddOutputCache` to specify caching configuration that applies to multiple endpoints. A policy can be selected for specific endpoints, while a base policy provides default caching configuration for a collection of endpoints.

The following highlighted code configures caching for all of the app's endpoints, with expiration time of 10 seconds. If an expiration time isn't specified, it defaults to one minute.

C#

```
builder.Services.AddOutputCache(options =>  
{  
    options.AddBasePolicy(builder =>  
        builder.Expire(TimeSpan.FromSeconds(10)));  
    options.AddPolicy("Expire20", builder =>  
        builder.Expire(TimeSpan.FromSeconds(20)));  
    options.AddPolicy("Expire30", builder =>  
        builder.Expire(TimeSpan.FromSeconds(30)));  
});
```

The following highlighted code creates two policies, each specifying a different expiration time. Selected endpoints can use the 20-second expiration, and others can use the 30-second expiration.

C#

```
builder.Services.AddOutputCache(options =>  
{  
    options.AddBasePolicy(builder =>  
        builder.Expire(TimeSpan.FromSeconds(10)));  
    options.AddPolicy("Expire20", builder =>  
        builder.Expire(TimeSpan.FromSeconds(20)));  
    options.AddPolicy("Expire30", builder =>  
        builder.Expire(TimeSpan.FromSeconds(30)));  
});
```


You can select a policy for an endpoint when calling the `CacheOutput` method or using the `[OutputCache]` attribute.

In a minimal API app, the following code configures one endpoint with a 20-second expiration and one with a 30-second expiration:

C#

```
app.MapGet("/20", Gravatar.WriteGravatar).CacheOutput("Expire20");
app.MapGet("/30", [OutputCache(PolicyName = "Expire30")] (context) =>
    Gravatar.WriteGravatar(context));
```

For apps with controllers, apply the `[OutputCache]` attribute to the action method to select a policy:

C#

```
[ApiController]
[Route("/[controller]")]
[OutputCache(PolicyName = "Expire20")]
public class Expire20Controller : ControllerBase
{
    public async Task GetAsync()
    {
        await Gravatar.WriteGravatar(HttpContext);
    }
}
```

For Razor Pages apps, apply the attribute to the Razor page class.

Default output caching policy

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.

The following code applies all of the default caching rules to all of an app's endpoints:

C#

```
builder.Services.AddOutputCache(options =>
{
```

```
options.AddBasePolicy(builder => builder.Cache());
});
```

Override the default policy

The following code shows how to override the default rules. The highlighted lines in the following custom policy code enable caching for HTTP POST methods and HTTP 301 responses:

C#

```
using Microsoft.AspNetCore.OutputCaching;
using Microsoft.Extensions.Primitives;

namespace OCMinimal;

public sealed class MyCustomPolicy : IOutputCachePolicy
{
    public static readonly MyCustomPolicy Instance = new();

    private MyCustomPolicy()
    {
    }

    ValueTask IOutputCachePolicy.CacheRequestAsync(
        OutputCacheContext context,
        CancellationToken cancellationToken)
    {
        var attemptOutputCaching = AttemptOutputCaching(context);
        context.EnableOutputCaching = true;
        context.AllowCacheLookup = attemptOutputCaching;
        context.AllowCacheStorage = attemptOutputCaching;
        context.AllowLocking = true;

        // Vary by any query by default
        context.CacheVaryByRules.QueryKeys = "*";

        return ValueTask.CompletedTask;
    }

    ValueTask IOutputCachePolicy.ServeFromCacheAsync(
        OutputCacheContext context, CancellationToken cancellationToken)
    {
        return ValueTask.CompletedTask;
    }

    ValueTask IOutputCachePolicy.ServeResponseAsync(
        OutputCacheContext context, CancellationToken cancellationToken)
    {
        var response = context.HttpContext.Response;
```

```

// Verify existence of cookie headers
if (!StringValues.IsNullOrEmpty(response.Headers.SetCookie))
{
    context.AllowCacheStorage = false;
    return ValueTask.CompletedTask;
}

// Check response code
if (response.StatusCode != StatusCodes.Status200OK &&
    response.StatusCode != StatusCodes.Status301MovedPermanently)
{
    context.AllowCacheStorage = false;
    return ValueTask.CompletedTask;
}

return ValueTask.CompletedTask;
}

private static bool AttemptOutputCaching(OutputCacheContext context)
{
    // Check if the current request fulfills the requirements
    // to be cached
    var request = context.HttpContext.Request;

    // Verify the method
    if (!HttpMethods.IsGet(request.Method) &&
        !HttpMethods.IsHead(request.Method) &&
        !HttpMethods.IsPost(request.Method))
    {
        return false;
    }

    // Verify existence of authorization headers
    if (!StringValues.IsNullOrEmpty(request.Headers.Authorization) ||
        request.HttpContext.User?.Identity?.IsAuthenticated == true)
    {
        return false;
    }

    return true;
}
}

```

To use this custom policy, create a named policy:

C#

```

builder.Services.AddOutputCache(options =>
{
    options.AddPolicy("CachePost", MyCustomPolicy.Instance);
});

```

And select the named policy for an endpoint. The following code selects the custom policy for an endpoint in a minimal API app:

C#

```
app.MapPost("/cachedpost", Gravatar.WriteGravatar)
    .CacheOutput("CachePost");
```

The following code does the same for a controller action:

C#

```
[ApiController]
[Route("/[controller]")]
[OutputCache(PolicyName = "CachePost")]
public class PostController : ControllerBase
{
    public async Task GetAsync()
    {
        await Gravatar.WriteGravatar(HttpContext);
    }
}
```

Alternative default policy override

Alternatively, use Dependency Injection (DI) to initialize an instance, with the following changes to the custom policy class:

- A public constructor instead of a private constructor.
- Eliminate the `Instance` property in the custom policy class.

For example:

C#

```
public sealed class MyCustomPolicy2 : IOutputCachePolicy
{
    public MyCustomPolicy2()
    {
    }
}
```

The remainder of the class is the same as shown previously. Add the custom policy as shown in the following example:

C#

```
builder.Services.AddOutputCache(options =>
{
    options.AddPolicy("CachePost", builder =>
        builder.AddPolicy<MyCustomPolicy2>(), true);
});
```

The preceding code uses DI to create the instance of the custom policy class. Any public arguments in the constructor are resolved.

When using a custom policy as a base policy, don't call `OutputCache()` (with no arguments) or use the `[OutputCache]` attribute on any endpoint that the base policy should apply to. Calling `OutputCache()` or using the attribute adds the default policy to the endpoint.

Specify the cache key

By default, every part of the URL is included as the key to a cache entry, that is, the scheme, host, port, path, and query string. However, you might want to explicitly control the cache key. For example, suppose you have an endpoint that returns a unique response only for each unique value of the `culture` query string. Variation in other parts of the URL, such as other query strings, shouldn't result in different cache entries. You can specify such rules in a policy, as shown in the following highlighted code:

C#

```
builder.Services.AddOutputCache(options =>
{
    options.AddBasePolicy(builder => builder
        .With(c => c.HttpContext.Request.Path.StartsWithSegments("/blog"))
        .Tag("tag-blog"));
    options.AddBasePolicy(builder => builder.Tag("tag-all"));
    options.AddPolicy("Query", builder =>
        builder.SetVaryByQuery("culture"));
    options.AddPolicy("NoCache", builder => builder.NoCache());
    options.AddPolicy("NoLock", builder => builder.SetLocking(false));
});
```

You can then select the `VaryByQuery` policy for an endpoint. In a minimal API app, the following code selects the `VaryByQuery` policy for an endpoint that returns a unique response only for each unique value of the `culture` query string:

C#

```
app.MapGet("/query", Gravatar.WriteGravatar).CacheOutput("Query");
```

The following code does the same for a controller action:

C#

```
[ApiController]
[Route("/[controller]")]
[OutputCache(PolicyName = "Query")]
public class QueryController : ControllerBase
{
    public async Task GetAsync()
    {
        await Gravatar.WriteGravatar(HttpContext);
    }
}
```

Here are some of the options for controlling the cache key:

- [SetVaryByQuery](#) - Specify one or more query string names to add to the cache key.
- [SetVaryByHeader](#) - Specify one or more HTTP headers to add to the cache key.
- [VaryByValue](#) - Specify a value to add to the cache key. The following example uses a value that indicates whether the current server time in seconds is odd or even. A new response is generated only when the number of seconds goes from odd to even or even to odd.

C#

```
builder.Services.AddOutputCache(options =>
{
    options.AddBasePolicy(builder => builder
        .With(c =>
            c.HttpContext.Request.Path.StartsWithSegments("/blog"))
        .Tag("tag-blog"));
    options.AddBasePolicy(builder => builder.Tag("tag-all"));
    options.AddPolicy("Query", builder =>
        builder.SetVaryByQuery("culture"));
    options.AddPolicy("NoCache", builder => builder.NoCache());
    options.AddPolicy("NoLock", builder => builder.SetLocking(false));
    options.AddPolicy("VaryByValue", builder =>
        builder.VaryByValue((context) =>
            new KeyValuePair<string, string>(
                "time", (DateTime.Now.Second % 2)
                    .ToString(CultureInfo.InvariantCulture))));
});
```

Use [OutputCacheOptions.UseCaseSensitivePaths](#) to specify that the path part of the key is case sensitive. The default is case insensitive.

For more options, see the [OutputCachePolicyBuilder](#) class.

Cache revalidation

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

The following code illustrates the use of an [Etag](#) header to enable cache revalidation. If the client sends an [If-None-Match](#) header with the etag value of an earlier response, and the cache entry is fresh, the server returns `304 Not Modified` instead of the full response. Here's how to set the etag value in a policy, in a Minimal API app:

C#

```
app.MapGet("/etag", async (context) =>
{
    var etag = $"\"{Guid.NewGuid():n}\"";
    context.Response.Headers.ETag = etag;
    await Gravatar.WriteGravatar(context);

}).CacheOutput();
```

And here's how to set the etag value in a controller-based API:

C#

```
[ApiController]
[Route("/[controller]")]
[OutputCache]
public class EtagController : ControllerBase
{
    public async Task GetAsync()
    {
        var etag = $"\"{Guid.NewGuid():n}\"";
        HttpContext.Response.Headers.ETag = etag;
        await Gravatar.WriteGravatar(HttpContext);
    }
}
```

Another way to do cache revalidation is to check the date of the cache entry creation compared to the date requested by the client. When the request header `If-Modified-`

Since `is provided`, output caching returns 304 if the cached entry is older and isn't expired.

Cache revalidation is automatic in response to these headers sent from the client. No special configuration is required on the server to enable this behavior, aside from enabling output caching.

Use tags to evict cache entries

You can use tags to identify a group of endpoints and evict all cache entries for the group. For example, the following minimal API code creates a pair of endpoints whose URLs begin with "blog", and tags them "tag-blog":

C#

```
app.MapGet("/blog", Gravatar.WriteGravatar)
    .CacheOutput(builder => builder.Tag("tag-blog"));
app.MapGet("/blog/post/{id}", Gravatar.WriteGravatar)
    .CacheOutput(builder => builder.Tag("tag-blog"));
```

The following code shows how to assign tags to an endpoint in a controller-based API:

C#

```
[ApiController]
[Route("/[controller]")]
[OutputCache(Tags = new[] { "tag-blog", "tag-all" })]
public class TagEndpointController : ControllerBase
{
    public async Task GetAsync()
    {
        await Gravatar.WriteGravatar(HttpContext);
    }
}
```

An alternative way to assign tags for endpoints with routes that begin with `blog` is to define a base policy that applies to all endpoints with that route. The following code shows how to do that:

C#

```
builder.Services.AddOutputCache(options =>
{
    options.AddBasePolicy(builder => builder
        .With(c => c.HttpContext.Request.Path.StartsWithSegments("/blog"))
        .Tag("tag-blog"));
```



```

options.AddBasePolicy(builder => builder.Tag("tag-all"));
options.AddPolicy("Query", builder =>
builder.SetVaryByQuery("culture"));
options.AddPolicy("NoCache", builder => builder.NoCache());
options.AddPolicy("NoLock", builder => builder.SetLocking(false));
});

```

Another alternative for minimal API apps is to call `MapGroup`:

C#

```

var blog = app.MapGroup("blog")
    .CacheOutput(builder => builder.Tag("tag-blog"));
blog.MapGet("/", Gravatar.WriteGravatar);
blog.MapGet("/post/{id}", Gravatar.WriteGravatar);

```

In the preceding tag assignment examples, both endpoints are identified by the `tag-blog` tag. You can then evict the cache entries for those endpoints with a single statement that references that tag:

C#

```

app.MapPost("/purge/{tag}", async (IOutputCacheStore cache, string tag) =>
{
    await cache.EvictByTagAsync(tag, default);
});

```

With this code, an HTTP POST request sent to `https://localhost:<port>/purge/tag-blog` evicts cache entries for these endpoints.

You might want a way to evict all cache entries for all endpoints. To do that, create a base policy for all endpoints as the following code does:

C#

```

builder.Services.AddOutputCache(options =>
{
    options.AddBasePolicy(builder => builder
        .With(c => c.HttpContext.Request.Path.StartsWithSegments("/blog"))
        .Tag("tag-blog"));
    options.AddBasePolicy(builder => builder.Tag("tag-all"));
    options.AddPolicy("Query", builder =>
builder.SetVaryByQuery("culture"));
    options.AddPolicy("NoCache", builder => builder.NoCache());
    options.AddPolicy("NoLock", builder => builder.SetLocking(false));
});

```

This base policy enables you to use the "tag-all" tag to evict everything in cache.

Disable resource locking

By default, resource locking is enabled to mitigate the risk of [cache stampede and thundering herd](#) [↗]. For more information, see [Output Caching](#).

To disable resource locking, call `SetLocking(false)` while creating a policy, as shown in the following example:

C#

```
builder.Services.AddOutputCache(options =>
{
    options.AddBasePolicy(builder => builder
        .With(c => c.HttpContext.Request.Path.StartsWithSegments("/blog"))
        .Tag("tag-blog"));
    options.AddBasePolicy(builder => builder.Tag("tag-all"));
    options.AddPolicy("Query", builder =>
        builder.SetVaryByQuery("culture"));
    options.AddPolicy("NoCache", builder => builder.NoCache());
    options.AddPolicy("NoLock", builder => builder.SetLocking(false));
});
```

The following example selects the no-locking policy for an endpoint in a minimal API app:

C#

```
app.MapGet("/no-lock", Gravatar.WriteGravatar)
    .CacheOutput("NoLock");
```

In a controller-based API, use the attribute to select the policy:

C#

```
[ApiController]
[Route("/[controller]")]
[OutputCache(PolicyName = "NoLock")]
public class NoLockController : ControllerBase
{
    public async Task GetAsync()
    {
        await Gravatar.WriteGravatar(HttpContext);
    }
}
```

Limits

The following properties of [OutputCacheOptions](#) let you configure limits that apply to all endpoints:

- [SizeLimit](#) - Maximum size of cache storage. When this limit is reached, no new responses are cached until older entries are evicted. Default value is 100 MB.
- [MaximumBodySize](#) - If the response body exceeds this limit, it isn't cached. Default value is 64 MB.
- [DefaultExpirationTimeSpan](#) - The expiration time duration that applies when not specified by a policy. Default value is 60 seconds.

Cache storage

[IOutputCacheStore](#) is used for storage. By default it's used with [MemoryCache](#). Cached responses are stored in-process, so each server has a separate cache that is lost whenever the server process is restarted.

Redis cache

An alternative is to use [Redis](#) cache. Redis cache provides consistency between server nodes via a shared cache that outlives individual server processes. To use Redis for output caching:

- Install the [Microsoft.AspNetCore.OutputCaching.StackExchangeRedis](#) NuGet package.
- Call `builder.Services.AddStackExchangeRedisOutputCache` (not `AddStackExchangeRedisCache`), and provide a connection string that points to a Redis server.

For example:

```
C#

builder.Services.AddStackExchangeRedisOutputCache(options =>
{
    options.Configuration =
        builder.Configuration.GetConnectionString("MyRedisConStr");
    options.InstanceName = "SampleInstance";
});

builder.Services.AddOutputCache(options =>
{
```

```
options.AddBasePolicy(builder =>
    builder.Expire(TimeSpan.FromSeconds(10)));
});
```

- [options.Configuration](#) - A connection string to an on-premises Redis server or to a hosted offering such as [Azure Cache for Redis](#). For example,

```
<instance_name>.redis.cache.windows.net:6380,password=
```

```
<password>,ssl=True,abortConnect=False
```

 for Azure cache for Redis.

- [options.InstanceName](#) - Optional, specifies a logical partition for the cache.

The configuration options are identical to the [Redis-based distributed caching options](#).

IDistributedCache not recommended

We don't recommend [IDistributedCache](#) for use with output caching. `IDistributedCache` doesn't have atomic features, which are required for tagging. We recommend that you use the built-in support for Redis or create custom [IOutputCacheStore](#) implementations by using direct dependencies on the underlying storage mechanism.

See also

- [Overview of caching in ASP.NET Core](#)
- [ASP.NET Core Middleware](#)
- [OutputCacheOptions](#)
- [OutputCachePolicyBuilder](#)

Response caching in ASP.NET Core

Article • 04/10/2024

By [Rick Anderson](#) and [Kirk Larkin](#)

[View or download sample code](#) (how to download)

Response caching reduces the number of requests a client or proxy makes to a web server. Response caching also reduces the amount of work the web server performs to generate a response. Response caching is set in headers.

The [ResponseCache attribute](#) sets response caching headers. Clients and intermediate proxies should honor the headers for caching responses under [RFC 9111: HTTP Caching](#).

For server-side caching that follows the HTTP 1.1 Caching specification, use [Response Caching Middleware](#). The middleware can use the [ResponseCacheAttribute](#) properties to influence server-side caching behavior.

The Response caching middleware:

- Enables caching server responses based on [HTTP cache headers](#). Implements the standard HTTP caching semantics. Caches based on HTTP cache headers like proxies do.
- Is typically not beneficial for UI apps such as Razor Pages because browsers generally set request headers that prevent caching. [Output caching](#), which is available in ASP.NET Core 7.0 and later, benefits UI apps. With output caching, configuration decides what should be cached independently of HTTP headers.
- May be beneficial for public GET or HEAD API requests from clients where the [Conditions for caching](#) are met.

To test response caching, use [Fiddler](#), or another tool that can explicitly set request headers. Setting headers explicitly is preferred for testing caching. For more information, see [Troubleshooting](#).

HTTP-based response caching

[RFC 9111: HTTP Caching](#) describes how Internet caches should behave. The primary HTTP header used for caching is [Cache-Control](#), which is used to specify cache directives. The directives control caching behavior as requests make their way from clients to servers and as responses make their way from servers back to clients. Requests

and responses move through proxy servers, and proxy servers must also conform to the HTTP 1.1 Caching specification.

Common `Cache-Control` directives are shown in the following table.

[Expand table](#)

Directive	Action
public	A cache may store the response.
private	The response must not be stored by a shared cache. A private cache may store and reuse the response.
max-age	The client doesn't accept a response whose age is greater than the specified number of seconds. Examples: <code>max-age=60</code> (60 seconds), <code>max-age=2592000</code> (1 month)
no-cache	On requests: A cache must not use a stored response to satisfy the request. The origin server regenerates the response for the client, and the middleware updates the stored response in its cache. On responses: The response must not be used for a subsequent request without validation on the origin server.
no-store	On requests: A cache must not store the request. On responses: A cache must not store any part of the response.

Other cache headers that play a role in caching are shown in the following table.

[Expand table](#)

Header	Function
Age	An estimate of the amount of time in seconds since the response was generated or successfully validated at the origin server.
Expires	The time after which the response is considered stale.
Pragma	Exists for backwards compatibility with HTTP/1.0 caches for setting <code>no-cache</code> behavior. If the <code>Cache-Control</code> header is present, the <code>Pragma</code> header is ignored.
Vary	Specifies that a cached response must not be sent unless all of the <code>Vary</code> header fields match in both the cached response's original request and the new request.

HTTP-based caching respects request `Cache-Control` directives

[RFC 9111: HTTP Caching \(Section 5.2. Cache-Control\)](#) [↗] requires a cache to honor a valid `Cache-Control` header sent by the client. A client can make requests with a `no-cache` header value and force the server to generate a new response for every request.

Always honoring client `Cache-Control` request headers makes sense if you consider the goal of HTTP caching. Under the official specification, caching is meant to reduce the latency and network overhead of satisfying requests across a network of clients, proxies, and servers. It isn't necessarily a way to control the load on an origin server.

There's no developer control over this caching behavior when using the [Response Caching Middleware](#) because the middleware adheres to the official caching specification. Support for *output caching* to better control server load was added in .NET 7. For more information, see [Output caching](#).

ResponseCache attribute

The [ResponseCacheAttribute](#) specifies the parameters necessary for setting appropriate headers in response caching.

Warning

Disable caching for content that contains information for authenticated clients. Caching should only be enabled for content that doesn't change based on a user's identity or whether a user is signed in.

[VaryByQueryKeys](#) varies the stored response by the values of the given list of query keys. When a single value of `*` is provided, the middleware varies responses by all request query string parameters.

[Response Caching Middleware](#) must be enabled to set the [VaryByQueryKeys](#) property. Otherwise, a runtime exception is thrown. There isn't a corresponding HTTP header for the [VaryByQueryKeys](#) property. The property is an HTTP feature handled by Response Caching Middleware. For the middleware to serve a cached response, the query string and query string value must match a previous request. For example, consider the sequence of requests and results shown in the following table:

 Expand table

Request	Returned from
<code>http://example.com?key1=value1</code>	Server

Request	Returned from
<code>http://example.com?key1=value1</code>	Middleware
<code>http://example.com?key1=NewValue</code>	Server

The first request is returned by the server and cached in middleware. The second request is returned by middleware because the query string matches the previous request. The third request isn't in the middleware cache because the query string value doesn't match a previous request.

The [ResponseCacheAttribute](#) is used to configure and create (via [IFilterFactory](#)) a `Microsoft.AspNetCore.Mvc.Internal.ResponseCacheFilter`. The `ResponseCacheFilter` performs the work of updating the appropriate HTTP headers and features of the response. The filter:

- Removes any existing headers for `Vary`, `Cache-Control`, and `Pragma`.
- Writes out the appropriate headers based on the properties set in the [ResponseCacheAttribute](#).
- Updates the response caching HTTP feature if [VaryByQueryKeys](#) is set.

Vary

This header is only written when the [VaryByHeader](#) property is set. The property set to the `Vary` property's value. The following sample uses the [VaryByHeader](#) property:

C#

```
[ApiController]
public class TimeController : ControllerBase
{
    [Route("api/[controller]")]
    [HttpGet]
    [ResponseCache(VaryByHeader = "User-Agent", Duration = 30)]
    public ContentResult GetTime() => Content(
        DateTime.Now.Millisecond.ToString());
}
```

View the response headers with Fiddler or another tool. The response headers include:

text

```
Cache-Control: public,max-age=30
Vary: User-Agent
```


The preceding code requires adding the Response Caching Middleware services [AddResponseCaching](#) to the service collection and configures the app to use the middleware with the [UseResponseCaching](#) extension method.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddResponseCaching();

var app = builder.Build();

app.UseHttpsRedirection();

// UseCors must be called before UseResponseCaching
//app.UseCors();

app.UseResponseCaching();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

NoStore and Location.None

[NoStore](#) overrides most of the other properties. When this property is set to `true`, the `Cache-Control` header is set to `no-store`. If [Location](#) is set to `None`:

- `Cache-Control` is set to `no-store, no-cache`.
- `Pragma` is set to `no-cache`.

If [NoStore](#) is `false` and [Location](#) is `None`, `Cache-Control`, and `Pragma` are set to `no-cache`.

[NoStore](#) is typically set to `true` for error pages. The following produces response headers that instruct the client not to store the response.

C#

```
[Route("api/[controller]/ticks")]
[HttpGet]
[ResponseCache(Location = ResponseCacheLocation.None, NoStore = true)]
public ContentResult GetTimeTicks() => Content(
    DateTime.Now.Ticks.ToString());
```

The preceding code includes the following headers in the response:

```
Cache-Control: no-store,no-cache  
Pragma: no-cache
```

To apply the [ResponseCacheAttribute](#) to all of the app's MVC controller or Razor Pages page responses, add it with an [MVC filter](#) or [Razor Pages filter](#).

In an MVC app:

```
C#  
  
builder.Services.AddControllersWithViews().AddMvcOptions(options =>  
    options.Filters.Add(  
        new ResponseCacheAttribute  
        {  
            NoStore = true,  
            Location = ResponseCacheLocation.None  
        }  
    ));
```

For an approach that applies to Razor Pages apps, see [Adding ResponseCacheAttribute to MVC global filter list does not apply to Razor Pages \(dotnet/aspnetcore #18890\)](#) [↗](#). The example provided in the issue comment was written for apps targeting ASP.NET Core prior to the release of [Minimal APIs](#) at 6.0. For 6.0 or later apps, change the service registration in the example to `builder.Services.AddSingleton...` for `Program.cs`.

Location and Duration

To enable caching, [Duration](#) must be set to a positive value and [Location](#) must be either `Any` (the default) or `Client`. The framework sets the `Cache-Control` header to the location value followed by the `max-age` of the response.

[Location](#)'s options of `Any` and `Client` translate into `Cache-Control` header values of `public` and `private`, respectively. As noted in the [NoStore and Location.None](#) section, setting [Location](#) to `None` sets both `Cache-Control` and `Pragma` headers to `no-cache`.

`Location.Any` (`Cache-Control` set to `public`) indicates that the *client or any intermediate proxy* may cache the value, including [Response Caching Middleware](#).

`Location.Client` (`Cache-Control` set to `private`) indicates that *only the client* may cache the value. No intermediate cache should cache the value, including [Response Caching Middleware](#).

Cache control headers provide guidance to clients and intermediary proxies when and how to cache responses. There's no guarantee that clients and proxies will honor [RFC 9111: HTTP Caching](#). [Response Caching Middleware](#) always follows the caching rules laid out by the specification.

The following example shows the headers produced by setting [Duration](#) and leaving the default [Location](#) value:

C#

```
[Route("api/[controller]/ms")]
[HttpGet]
[ResponseCache(Duration = 10, Location = ResponseCacheLocation.Any, NoStore = false)]
public ContentResult GetTimeMS() => Content(
    DateTime.Now.Millisecond.ToString());
```

The preceding code includes the following headers in the response:

```
Cache-Control: public,max-age=10
```

Cache profiles

Instead of duplicating response cache settings on many controller action attributes, cache profiles can be configured as options when setting up MVC/Razor Pages. Values found in a referenced cache profile are used as the defaults by the [ResponseCacheAttribute](#) and are overridden by any properties specified on the attribute.

The following example shows a 30 second cache profile:

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);
```

```

builder.Services.AddResponseCaching();
builder.Services.AddControllers(options =>
{
    options.CacheProfiles.Add("Default30",
        new CacheProfile()
        {
            Duration = 30
        });
});

var app = builder.Build();

app.UseHttpsRedirection();

// UseCors must be called before UseResponseCaching
//app.UseCors();

app.UseResponseCaching();

app.UseAuthorization();

app.MapControllers();

app.Run();

```

The following code references the `Default30` cache profile:

```

C#

[ApiController]
[ResponseCache(CacheProfileName = "Default30")]
public class Time2Controller : ControllerBase
{
    [Route("api/[controller]")]
    [HttpGet]
    public ContentResult GetTime() => Content(
        DateTime.Now.Millisecond.ToString());

    [Route("api/[controller]/ticks")]
    [HttpGet]
    public ContentResult GetTimeTicks() => Content(
        DateTime.Now.Ticks.ToString());
}

```

The resulting header response by the `Default30` cache profile includes:

```

Cache-Control: public,max-age=30

```

The `[ResponseCache]` attribute can be applied to:

- Razor Pages: Attributes can't be applied to handler methods. Browsers used with UI apps prevent response caching.
- MVC controllers.
- MVC action methods: Method-level attributes override the settings specified in class-level attributes.

The following code applies the `[ResponseCache]` attribute at the controller level and method level:

```
C#

[ApiController]
[ResponseCache(VaryByHeader = "User-Agent", Duration = 30)]
public class Time4Controller : ControllerBase
{
    [Route("api/[controller]")]
    [HttpGet]
    public ContentResult GetTime() => Content(
        DateTime.Now.Millisecond.ToString());

    [Route("api/[controller]/ticks")]
    [HttpGet]
    public ContentResult GetTimeTicks() => Content(
        DateTime.Now.Ticks.ToString());

    [Route("api/[controller]/ms")]
    [HttpGet]
    [ResponseCache(Duration = 10, Location = ResponseCacheLocation.Any,
        NoStore = false)]
    public ContentResult GetTimeMS() => Content(
        DateTime.Now.Millisecond.ToString());
}
```

Additional resources

- [Storing Responses in Caches](#)
- [Cache-Control](#)
- [Cache in-memory in ASP.NET Core](#)
- [Distributed caching in ASP.NET Core](#)
- [Detect changes with change tokens in ASP.NET Core](#)
- [Response Caching Middleware in ASP.NET Core](#)
- [Cache Tag Helper in ASP.NET Core MVC](#)
- [Distributed Cache Tag Helper in ASP.NET Core](#)

Response Caching Middleware in ASP.NET Core

Article • 07/16/2024

By [John Luo](#) and [Rick Anderson](#)

This article explains how to configure [Response Caching Middleware](#) in an ASP.NET Core app. The middleware determines when responses are cacheable, stores responses, and serves responses from cache. For an introduction to HTTP caching and the [\[ResponseCache\]](#) attribute, see [Response Caching](#).

The Response caching middleware:

- Enables caching server responses based on [HTTP cache headers](#). Implements the standard HTTP caching semantics. Caches based on HTTP cache headers like proxies do.
- Is typically not beneficial for UI apps such as Razor Pages because browsers generally set request headers that prevent caching. [Output caching](#), which is available in ASP.NET Core 7.0 and later, benefits UI apps. With output caching, configuration decides what should be cached independently of HTTP headers.
- May be beneficial for public GET or HEAD API requests from clients where the [Conditions for caching](#) are met.

To test response caching, use [Fiddler](#), or another tool that can explicitly set request headers. Setting headers explicitly is preferred for testing caching. For more information, see [Troubleshooting](#).

Configuration

In `Program.cs`, add the Response Caching Middleware services [AddResponseCaching](#) to the service collection and configure the app to use the middleware with the [UseResponseCaching](#) extension method. `UseResponseCaching` adds the middleware to the request processing pipeline:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCaching();

var app = builder.Build();
```

```
app.UseHttpsRedirection();

// UseCors must be called before UseResponseCaching
//app.UseCors();

app.UseResponseCaching();
```

Warning

UseCors must be called before UseResponseCaching when using CORS middleware.

The sample app adds headers to control caching on subsequent requests:

- [Cache-Control](#): Caches cacheable responses for up to 10 seconds.
- [Vary](#): Configures the middleware to serve a cached response only if the [Accept-Encoding](#) header of subsequent requests matches that of the original request.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCaching();

var app = builder.Build();

app.UseHttpsRedirection();

// UseCors must be called before UseResponseCaching
//app.UseCors();

app.UseResponseCaching();

app.Use(async (context, next) =>
{
    context.Response.GetTypedHeaders().CacheControl =
        new Microsoft.Net.Http.Headers.CacheControlHeaderValue()
        {
            Public = true,
            MaxAge = TimeSpan.FromSeconds(10)
        };
    context.Response.Headers[Microsoft.Net.Http.Headers.HeaderNames.Vary] =
        new string[] { "Accept-Encoding" };

    await next();
});

app.MapGet("/", () => DateTime.Now.Millisecond);
```

```
app.Run();
```

The preceding headers are not written to the response and are overridden when a controller, action, or Razor Page:

- Has a [\[ResponseCache\]](#) attribute. This applies even if a property isn't set. For example, omitting the [VaryByHeader](#) property will cause the corresponding header to be removed from the response.

Response Caching Middleware only caches server responses that result in a 200 (OK) status code. Any other responses, including [error pages](#), are ignored by the middleware.

Warning

Responses containing content for authenticated clients must be marked as not cacheable to prevent the middleware from storing and serving those responses. See [Conditions for caching](#) for details on how the middleware determines if a response is cacheable.

The preceding code typically doesn't return a cached value to a browser. Use [Fiddler](#) or another tool that can explicitly set request headers and is preferred for testing caching. For more information, see [Troubleshooting](#) in this article.

Options

Response caching options are shown in the following table.

 Expand table

Option	Description
MaximumBodySize	The largest cacheable size for the response body in bytes. The default value is <code>64 * 1024 * 1024</code> (64 MB).
SizeLimit	The size limit for the response cache middleware in bytes. The default value is <code>100 * 1024 * 1024</code> (100 MB).
UseCaseSensitivePaths	Determines if responses are cached on case-sensitive paths. The default value is <code>false</code> .

The following example configures the middleware to:

- Cache responses with a body size smaller than or equal to 1,024 bytes.

- Store the responses by case-sensitive paths. For example, `/page1` and `/Page1` are stored separately.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCaching(options =>
{
    options.MaximumBodySize = 1024;
    options.UseCaseSensitivePaths = true;
});

var app = builder.Build();

app.UseHttpsRedirection();

// UseCors must be called before UseResponseCaching
//app.UseCors();

app.UseResponseCaching();

app.Use(async (context, next) =>
{
    context.Response.GetTypedHeaders().CacheControl =
        new Microsoft.Net.Http.Headers.CacheControlHeaderValue()
        {
            Public = true,
            MaxAge = TimeSpan.FromSeconds(10)
        };
    context.Response.Headers[Microsoft.Net.Http.Headers.HeaderNames.Vary] =
        new string[] { "Accept-Encoding" };

    await next(context);
});

app.MapGet("/", () => DateTime.Now.Millisecond);

app.Run();
```

VaryByQueryKeys

When using MVC, web API controllers, or Razor Pages page models, the [\[ResponseCache\]](#) attribute specifies the parameters necessary for setting the appropriate headers for response caching. The only parameter of the [\[ResponseCache\]](#) attribute that strictly requires the middleware is [VaryByQueryKeys](#), which doesn't correspond to an actual HTTP header. For more information, see [Response caching in ASP.NET Core](#).

When not using the `[ResponseCache]` attribute, response caching can be varied with `VaryByQueryKeys`. Use the [ResponseCachingFeature](#) directly from the [HttpContext.Features](#):

C#

```
var responseCachingFeature =  
context.HttpContext.Features.Get<IResponseCachingFeature>();  
  
if (responseCachingFeature != null)  
{  
    responseCachingFeature.VaryByQueryKeys = new[] { "MyKey" };  
}
```

Using a single value equal to `*` in `VaryByQueryKeys` varies the cache by all request query parameters.

HTTP headers used by Response Caching Middleware

The following table provides information on HTTP headers that affect response caching.

[Expand table](#)

Header	Details
Authorization	The response isn't cached if the header exists.
Cache-Control	<p>The middleware only considers caching responses marked with the <code>public</code> cache directive. Control caching with the following parameters:</p> <ul style="list-style-type: none">• max-age• max-stale[†]• min-fresh• must-revalidate• no-cache• no-store• only-if-cached• private• public• s-maxage• proxy-revalidate[‡] <p>[†]If no limit is specified to <code>max-stale</code>, the middleware takes no action.</p> <p>[‡]<code>proxy-revalidate</code> has the same effect as <code>must-revalidate</code>.</p>

Header	Details
	For more information, see RFC 9111: Request Directives ↗ .
Pragma	A <code>Pragma: no-cache</code> header in the request produces the same effect as <code>Cache-Control: no-cache</code> . This header is overridden by the relevant directives in the <code>Cache-Control</code> header, if present. Considered for backward compatibility with HTTP/1.0.
Set-Cookie	The response isn't cached if the header exists. Any middleware in the request processing pipeline that sets one or more cookies prevents the Response Caching Middleware from caching the response (for example, the cookie-based TempData provider).
Vary	The <code>Vary</code> header is used to vary the cached response by another header. For example, cache responses by encoding by including the <code>Vary: Accept-Encoding</code> header, which caches responses for requests with headers <code>Accept-Encoding: gzip</code> and <code>Accept-Encoding: text/plain</code> separately. A response with a header value of <code>*</code> is never stored.
Expires	A response deemed stale by this header isn't stored or retrieved unless overridden by other <code>Cache-Control</code> headers.
If-None-Match	The full response is served from cache if the value isn't <code>*</code> and the <code>ETag</code> of the response doesn't match any of the values provided. Otherwise, a 304 (Not Modified) response is served.
If-Modified-Since	If the <code>If-None-Match</code> header isn't present, a full response is served from cache if the cached response date is newer than the value provided. Otherwise, a <i>304 - Not Modified</i> response is served.
Date	When serving from cache, the <code>Date</code> header is set by the middleware if it wasn't provided on the original response.
Content-Length	When serving from cache, the <code>Content-Length</code> header is set by the middleware if it wasn't provided on the original response.
Age	The <code>Age</code> header sent in the original response is ignored. The middleware computes a new value when serving a cached response.

Caching respects request Cache-Control directives

The middleware respects the rules of [RFC 9111: HTTP Caching \(Section 5.2. Cache-Control\)](#) [↗](#). The rules require a cache to honor a valid `Cache-Control` header sent by the client. Under the specification, a client can make requests with a `no-cache` header value

and force the server to generate a new response for every request. Currently, there's no developer control over this caching behavior when using the middleware because the middleware adheres to the official caching specification.

For more control over caching behavior, explore other caching features of ASP.NET Core. See the following topics:

- [Cache in-memory in ASP.NET Core](#)
- [Distributed caching in ASP.NET Core](#)
- [Cache Tag Helper in ASP.NET Core MVC](#)
- [Distributed Cache Tag Helper in ASP.NET Core](#)

Troubleshooting

The [Response Caching Middleware](#) uses [IMemoryCache](#), which has a limited capacity. When the capacity is exceeded, the [memory cache is compacted](#).

If caching behavior isn't as expected, confirm that responses are cacheable and capable of being served from the cache. Examine the request's incoming headers and the response's outgoing headers. Enable [logging](#) to help with debugging.

When testing and troubleshooting caching behavior, a browser typically sets request headers that prevent caching. For example, a browser may set the `Cache-Control` header to `no-cache` or `max-age=0` when refreshing a page. [Fiddler](#) and other tools can explicitly set request headers and are preferred for testing caching.

Conditions for caching

- The request must result in a server response with a 200 (OK) status code.
- The request method must be GET or HEAD.
- Response Caching Middleware must be placed before middleware that require caching. For more information, see [ASP.NET Core Middleware](#).
- The `Authorization` header must not be present.
- `Cache-Control` header parameters must be valid, and the response must be marked `public` and not marked `private`.
- The `Pragma: no-cache` header must not be present if the `Cache-Control` header isn't present, as the `Cache-Control` header overrides the `Pragma` header when present.
- The `Set-Cookie` header must not be present.
- `Vary` header parameters must be valid and not equal to `*`.

- The `Content-Length` header value (if set) must match the size of the response body.
- The `IHttpSendFileFeature` isn't used.
- The response must not be stale as specified by the `Expires` header and the `max-age` and `s-maxage` cache directives.
- Response buffering must be successful. The size of the response must be smaller than the configured or default `SizeLimit`. The body size of the response must be smaller than the configured or default `MaximumBodySize`.
- The response must be cacheable according to [RFC 9111: HTTP Caching](#). For example, the `no-store` directive must not exist in request or response header fields. See [RFC 9111: HTTP Caching \(Section 3: Storing Responses in Caches\)](#) for details.

ⓘ Note

The Antiforgery system for generating secure tokens to prevent Cross-Site Request Forgery (CSRF) attacks sets the `Cache-Control` and `Pragma` headers to `no-cache` so that responses aren't cached. For information on how to disable antiforgery tokens for HTML form elements, see [Prevent Cross-Site Request Forgery \(XSRF/CSRF\) attacks in ASP.NET Core](#).

Additional resources

- [View or download sample code](#) (how to download)
- [GitHub source for IResponseCachingPolicyProvider](#)
- [GitHub source for IResponseCachingPolicyProvider](#)
- [App startup in ASP.NET Core](#)
- [ASP.NET Core Middleware](#)
- [Cache in-memory in ASP.NET Core](#)
- [Distributed caching in ASP.NET Core](#)
- [Detect changes with change tokens in ASP.NET Core](#)
- [Response caching in ASP.NET Core](#)
- [Cache Tag Helper in ASP.NET Core MVC](#)
- [Distributed Cache Tag Helper in ASP.NET Core](#)

Rate limiting middleware in ASP.NET Core

Article • 06/17/2024

By [Arvin Kahbazi](#), [Maarten Balliauw](#), and [Rick Anderson](#)

The `Microsoft.AspNetCore.RateLimiting` middleware provides rate limiting middleware. Apps configure rate limiting policies and then attach the policies to endpoints. Apps using rate limiting should be carefully load tested and reviewed before deploying. See [Testing endpoints with rate limiting](#) in this article for more information.

For an introduction to rate limiting, see [Rate limiting middleware](#).

Rate limiter algorithms

The `RateLimiterOptionsExtensions` class provides the following extension methods for rate limiting:

- [Fixed window](#)
- [Sliding window](#)
- [Token bucket](#)
- [Concurrency](#)

Fixed window limiter

The `AddFixedWindowLimiter` method uses a fixed time window to limit requests. When the time window expires, a new time window starts and the request limit is reset.

Consider the following code:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRateLimiter(_ => _
    .AddFixedWindowLimiter(policyName: "fixed", options =>
    {
        options.PermitLimit = 4;
        options.Window = TimeSpan.FromSeconds(12);
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
```

```

        options.QueueLimit = 2;
    }));

var app = builder.Build();

app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("0000");

app.MapGet("/", () => Results.Ok($"Hello {GetTicks()}"))
    .RequireRateLimiting("fixed");

app.Run();

```

The preceding code:

- Calls [AddRateLimiter](#) to add a rate limiting service to the service collection.
- Calls `AddFixedWindowLimiter` to create a fixed window limiter with a policy name of `"fixed"` and sets:
 - [PermitLimit](#) to 4 and the time [Window](#) to 12. A maximum of 4 requests per each 12-second window are allowed.
 - [QueueProcessingOrder](#) to [OldestFirst](#).
 - [QueueLimit](#) to 2.
- Calls [UseRateLimiter](#) to enable rate limiting.

Apps should use [Configuration](#) to set limiter options. The following code updates the preceding code using [MyRateLimitOptions](#) [↗](#) for configuration:

C#

```

using System.Threading.RateLimiting;
using Microsoft.AspNetCore.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);
builder.Services.Configure<MyRateLimitOptions>(
    builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit));

var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOptions);
var fixedPolicy = "fixed";

builder.Services.AddRateLimiter(_ => _
    .AddFixedWindowLimiter(policyName: fixedPolicy, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
    }

```

```

        options.QueueLimit = myOptions.QueueLimit;
    }));

    var app = builder.Build();

    app.UseRateLimiter();

    static string GetTicks() => (DateTime.Now.Ticks &
    0x11111).ToString("00000");

    app.MapGet("/", () => Results.Ok($"Fixed Window Limiter {GetTicks()}"))
        .RequireRateLimiting(fixedPolicy);

    app.Run();

```

`UseRateLimiter` must be called after `UseRouting` when rate limiting endpoint specific APIs are used. For example, if the `[EnableRateLimiting]` attribute is used, `UseRateLimiter` must be called after `UseRouting`. When calling only global limiters, `UseRateLimiter` can be called before `UseRouting`.

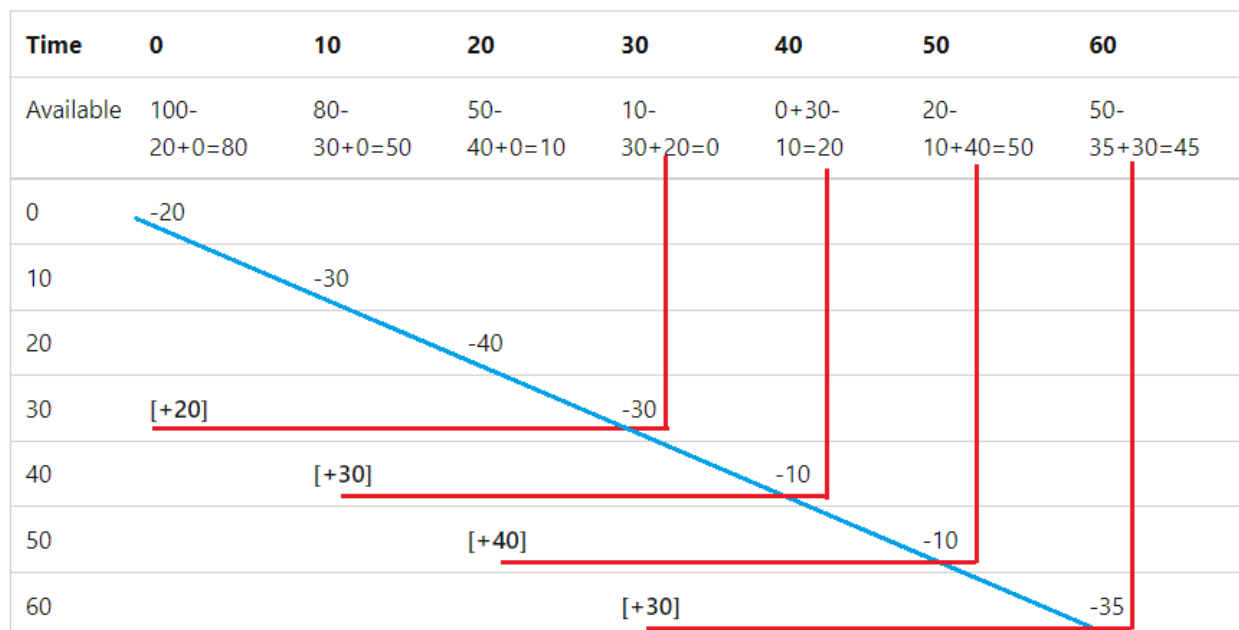
Sliding window limiter

A sliding window algorithm:

- Is similar to the fixed window limiter but adds segments per window. The window slides one segment each segment interval. The segment interval is (window time)/(segments per window).
- Limits the requests for a window to `permitLimit` requests.
- Each time window is divided in `n` segments per window.
- Requests taken from the expired time segment one window back (`n` segments prior to the current segment) are added to the current segment. We refer to the most expired time segment one window back as the expired segment.

Consider the following table that shows a sliding window limiter with a 30-second window, three segments per window, and a limit of 100 requests:

- The top row and first column shows the time segment.
- The second row shows the remaining requests available. The remaining requests are calculated as the available requests minus the processed requests plus the recycled requests.
- Requests at each time moves along the diagonal blue line.
- From time 30 on, the request taken from the expired time segment are added back to the request limit, as shown in the red lines.



The following table shows the data in the previous graph in a different format. The **Available** column shows the requests available from the previous segment (The **Carry over** from the previous row). The first row shows 100 available requests because there's no previous segment.

[Expand table](#)

Time	Available	Taken	Recycled from expired	Carry over
0	100	20	0	80
10	80	30	0	50
20	50	40	0	10
30	10	30	20	0
40	0	10	30	20
50	20	10	40	50
60	50	35	30	45

The following code uses the sliding window rate limiter:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);
```

```

var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOptions);
var slidingPolicy = "sliding";

builder.Services.AddRateLimiter(_ => _
    .AddSlidingWindowLimiter(policyName: slidingPolicy, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.SegmentsPerWindow = myOptions.SegmentsPerWindow;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var app = builder.Build();

app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
    0x11111).ToString("00000");

app.MapGet("/", () => Results.Ok($"Sliding Window Limiter {GetTicks()}"))
    .RequireRateLimiting(slidingPolicy);

app.Run();

```

Token bucket limiter

The token bucket limiter is similar to the sliding window limiter, but rather than adding back the requests taken from the expired segment, a fixed number of tokens are added each replenishment period. The tokens added each segment can't increase the available tokens to a number higher than the token bucket limit. The following table shows a token bucket limiter with a limit of 100 tokens and a 10-second replenishment period.

[Expand table](#)

Time	Available	Taken	Added	Carry over
0	100	20	0	80
10	80	10	20	90
20	90	5	15	100
30	100	30	20	90
40	90	6	16	100
50	100	40	20	80

Time	Available	Taken	Added	Carry over
60	80	50	20	50

The following code uses the token bucket limiter:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);

var tokenPolicy = "token";
var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOptions);

builder.Services.AddRateLimiter(_ => _
    .AddTokenBucketLimiter(policyName: tokenPolicy, options =>
    {
        options.TokenLimit = myOptions.TokenLimit;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
        options.ReplenishmentPeriod =
        TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod);
        options.TokensPerPeriod = myOptions.TokensPerPeriod;
        options.AutoReplenishment = myOptions.AutoReplenishment;
    }));

var app = builder.Build();

app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
    0x11111).ToString("00000");

app.MapGet("/", () => Results.Ok($"Token Limiter {GetTicks()}"))
    .RequireRateLimiting(tokenPolicy);

app.Run();
```

When [AutoReplenishment](#) is set to `true`, an internal timer replenishes the tokens every [ReplenishmentPeriod](#); when set to `false`, the app must call [TryReplenish](#) on the limiter.

Concurrency limiter

The concurrency limiter limits the number of concurrent requests. Each request reduces the concurrency limit by one. When a request completes, the limit is increased by one.

Unlike the other requests limiters that limit the total number of requests for a specified period, the concurrency limiter limits only the number of concurrent requests and doesn't cap the number of requests in a time period.

The following code uses the concurrency limiter:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);

var concurrencyPolicy = "Concurrency";
var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOptions);

builder.Services.AddRateLimiter(_ => _
    .AddConcurrencyLimiter(policyName: concurrencyPolicy, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var app = builder.Build();

app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
    0x11111).ToString("00000");

app.MapGet("/", async () =>
{
    await Task.Delay(500);
    return Results.Ok($"Concurrency Limiter {GetTicks()}");
}).RequireRateLimiting(concurrencyPolicy);

app.Run();
```

Create chained limiters

The [CreateChained](#) API allows passing in multiple [PartitionedRateLimiter](#) which are combined into one `PartitionedRateLimiter`. The combined limiter runs all the input limiters in sequence.

The following code uses `CreateChained`:

C#

```
using System.Globalization;
using System.Threading.RateLimiting;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRateLimiter(_ =>
{
    _.OnRejected = (context, _) =>
    {
        if (context.Lease.TryGetMetadata(MetadataName.RetryAfter, out var
retryAfter))
        {
            context.HttpContext.Response.Headers.RetryAfter =
                ((int)
retryAfter.TotalSeconds).ToString(NumberFormatInfo.InvariantInfo);
        }

        context.HttpContext.Response.StatusCode =
StatusCodes.Status429TooManyRequests;
        context.HttpContext.Response.WriteAsync("Too many requests. Please
try again later.");

        return new ValueTask();
    };
    _.GlobalLimiter = PartitionedRateLimiter.CreateChained(
        PartitionedRateLimiter.Create<HttpContext, string>(httpContext =>
        {
            var userAgent =
httpContext.Request.Headers.UserAgent.ToString();

            return RateLimitPartition.GetFixedWindowLimiter
                (userAgent, _ =>
                    new FixedWindowRateLimiterOptions
                    {
                        AutoReplenishment = true,
                        PermitLimit = 4,
                        Window = TimeSpan.FromSeconds(2)
                    });
        }),
        PartitionedRateLimiter.Create<HttpContext, string>(httpContext =>
        {
            var userAgent =
httpContext.Request.Headers.UserAgent.ToString();

            return RateLimitPartition.GetFixedWindowLimiter
                (userAgent, _ =>
                    new FixedWindowRateLimiterOptions
                    {
                        AutoReplenishment = true,
                        PermitLimit = 20,
                        Window = TimeSpan.FromSeconds(30)
                    });
        });
});
```

```

    }));
});

var app = builder.Build();
app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
    0x111111).ToString("00000");

app.MapGet("/", () => Results.Ok($"Hello {GetTicks()}"));

app.Run();

```

For more information, see the [CreateChained source code](#) ↗

EnableRateLimiting and DisableRateLimiting attributes

The [\[EnableRateLimiting\]](#) and [\[DisableRateLimiting\]](#) attributes can be applied to a Controller, action method, or Razor Page. For Razor Pages, the attribute must be applied to the Razor Page and not the page handlers. For example, [\[EnableRateLimiting\]](#) can't be applied to `OnGet`, `OnPost`, or any other page handler.

The [\[DisableRateLimiting\]](#) attribute **disables** rate limiting to the Controller, action method, or Razor Page regardless of named rate limiters or global limiters applied. For example, consider the following code which calls [RequireRateLimiting](#) to apply the `fixedPolicy` rate limiting to all controller endpoints:

```

C#

using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.Configure<MyRateLimitOptions>(
    builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit));

var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOptions);
var fixedPolicy = "fixed";

builder.Services.AddRateLimiter(_ => _

```

```

.AddFixedWindowLimiter(policyName: fixedPolicy, options =>
{
    options.PermitLimit = myOptions.PermitLimit;
    options.Window = TimeSpan.FromSeconds(myOptions.Window);
    options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
    options.QueueLimit = myOptions.QueueLimit;
}));

var slidingPolicy = "sliding";

builder.Services.AddRateLimiter(_ => _
.AddSlidingWindowLimiter(policyName: slidingPolicy, options =>
{
    options.PermitLimit = myOptions.SlidingPermitLimit;
    options.Window = TimeSpan.FromSeconds(myOptions.Window);
    options.SegmentsPerWindow = myOptions.SegmentsPerWindow;
    options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
    options.QueueLimit = myOptions.QueueLimit;
}));

var app = builder.Build();
app.UseRateLimiter();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.MapRazorPages().RequireRateLimiting(slidingPolicy);
app.MapDefaultControllerRoute().RequireRateLimiting(fixedPolicy);

app.Run();

```

In the following code, `[DisableRateLimiting]` disables rate limiting and overrides `[EnableRateLimiting("fixed")]` applied to the `Home2Controller` and `app.MapDefaultControllerRoute().RequireRateLimiting(fixedPolicy)` called in `Program.cs`:

C#

```

[EnableRateLimiting("fixed")]
public class Home2Controller : Controller
{
    private readonly ILogger<Home2Controller> _logger;

    public Home2Controller(ILogger<Home2Controller> logger)
    {
        _logger = logger;
    }
}

```

```

    }

    public ActionResult Index()
    {
        return View();
    }

    [EnableRateLimiting("sliding")]
    public ActionResult Privacy()
    {
        return View();
    }

    [DisableRateLimiting]
    public ActionResult NoLimit()
    {
        return View();
    }

    [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None,
    NoStore = true)]
    public IActionResult Error()
    {
        return View(new ErrorViewModel { RequestId = Activity.Current?.Id ??
    HttpContext.TraceIdentifier });
    }
}

```

In the preceding code, the `[EnableRateLimiting("sliding")]` is **not** applied to the `Privacy` action method because `Program.cs` called `app.MapDefaultControllerRoute().RequireRateLimiting(fixedPolicy).`

Consider the following code which doesn't call `RequireRateLimiting` on `MapRazorPages` or `MapDefaultControllerRoute`:

```

C#

using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.Configure<MyRateLimitOptions>(
    builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit));

var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti

```



```

ons);
var fixedPolicy = "fixed";

builder.Services.AddRateLimiter(_ => _
    .AddFixedWindowLimiter(policyName: fixedPolicy, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var slidingPolicy = "sliding";

builder.Services.AddRateLimiter(_ => _
    .AddSlidingWindowLimiter(policyName: slidingPolicy, options =>
    {
        options.PermitLimit = myOptions.SlidingPermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.SegmentsPerWindow = myOptions.SegmentsPerWindow;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var app = builder.Build();

app.UseRateLimiter();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.MapRazorPages();
app.MapDefaultControllerRoute(); // RequireRateLimiting not called

app.Run();

```

Consider the following controller:

C#

```

[EnableRateLimiting("fixed")]
public class Home2Controller : Controller
{
    private readonly ILogger<Home2Controller> _logger;

    public Home2Controller(ILogger<Home2Controller> logger)
    {

```

```

        _logger = logger;
    }

    public ActionResult Index()
    {
        return View();
    }

    [EnableRateLimiting("sliding")]
    public ActionResult Privacy()
    {
        return View();
    }

    [DisableRateLimiting]
    public ActionResult NoLimit()
    {
        return View();
    }

    [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None,
    NoStore = true)]
    public IActionResult Error()
    {
        return View(new ErrorViewModel { RequestId = Activity.Current?.Id ??
    HttpContext.TraceIdentifier });
    }
}

```

In the preceding controller:

- The "fixed" policy rate limiter is applied to all action methods that don't have `EnableRateLimiting` and `DisableRateLimiting` attributes.
- The "sliding" policy rate limiter is applied to the `Privacy` action.
- Rate limiting is disabled on the `NoLimit` action method.

Applying attributes to Razor Pages

For Razor Pages, the attribute must be applied to the Razor Page and not the page handlers. For example, `[EnableRateLimiting]` can't be applied to `OnGet`, `OnPost`, or any other page handler.

The `DisableRateLimiting` attribute disables rate limiting on a Razor Page.

`EnableRateLimiting` is only applied to a Razor Page if

`MapRazorPages().RequireRateLimiting(Policy)` has **not** been called.

Limiter algorithm comparison

The fixed, sliding, and token limiters all limit the maximum number of requests in a time period. The concurrency limiter limits only the number of concurrent requests and doesn't cap the number of requests in a time period. The cost of an endpoint should be considered when selecting a limiter. The cost of an endpoint includes the resources used, for example, time, data access, CPU, and I/O.

Rate limiter samples

The following samples aren't meant for production code but are examples on how to use the limiters.

Limiter with `OnRejected`, `RetryAfter`, and `GlobalLimiter`

The following sample:

- Creates a `RateLimiterOptions.OnRejected` callback that is called when a request exceeds the specified limit. `retryAfter` can be used with the [TokenBucketRateLimiter](#), [FixedWindowLimiter](#), and [SlidingWindowLimiter](#) because these algorithms are able to estimate when more permits will be added. The `ConcurrencyLimiter` has no way of calculating when permits will be available.
- Adds the following limiters:
 - A `SampleRateLimiterPolicy` which implements the `IRateLimiterPolicy<TPartitionKey>` interface. The `SampleRateLimiterPolicy` class is shown later in this article.
 - A `SlidingWindowLimiter`:
 - With a partition for each authenticated user.
 - One shared partition for all anonymous users.
 - A `GlobalLimiter` that is applied to all requests. The global limiter will be executed first, followed by the endpoint-specific limiter, if one exists. The `GlobalLimiter` creates a partition for each [IPAddress](#).

C#

```
// Preceding code removed for brevity.
using System.Globalization;
using System.Net;
using System.Threading.RateLimiting;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebRateLimitAuth;
using WebRateLimitAuth.Data;
using WebRateLimitAuth.Models;
```

```

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection") ??
    throw new InvalidOperationException("Connection string
'DefaultConnection' not found.");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.Configure<MyRateLimitOptions>(
    builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit));

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var userPolicyName = "user";
var helloPolicy = "hello";
var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOptions);

builder.Services.AddRateLimiter(limiterOptions =>
{
    limiterOptions.OnRejected = (context, cancellationToken) =>
    {
        if (context.Lease.TryGetMetadata(MetadataName.RetryAfter, out var
retryAfter))
        {
            context.HttpContext.Response.Headers.RetryAfter =
                ((int)
retryAfter.TotalSeconds).ToString(NumberFormatInfo.InvariantInfo);
        }

        context.HttpContext.Response.StatusCode =
StatusCodes.Status429TooManyRequests;
        context.HttpContext.RequestServices.GetService<ILoggerFactory>()?
            .CreateLogger("Microsoft.AspNetCore.RateLimitingMiddleware")
            .LogWarning("OnRejected: {GetUserEndPoint}",
GetUserEndPoint(context.HttpContext));

        return new ValueTask();
    }
});

limiterOptions.AddPolicy<string, SampleRateLimiterPolicy>(helloPolicy);
limiterOptions.AddPolicy(userPolicyName, context =>
{
    var username = "anonymous user";
    if (context.User.Identity?.IsAuthenticated is true)
    {

```

```

        username = context.User.ToString()!;
    }

    return RateLimitPartition.GetSlidingWindowLimiter(username,
        _ => new SlidingWindowRateLimiterOptions
        {
            PermitLimit = myOptions.PermitLimit,
            QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
            QueueLimit = myOptions.QueueLimit,
            Window = TimeSpan.FromSeconds(myOptions.Window),
            SegmentsPerWindow = myOptions.SegmentsPerWindow
        });

});

limiterOptions.GlobalLimiter =
PartitionedRateLimiter.Create<HttpContext, IPAddress>(context =>
{
    IPAddress? remoteIpAddress = context.Connection.RemoteIpAddress;

    if (!IPAddress.IsLoopback(remoteIpAddress!))
    {
        return RateLimitPartition.GetTokenBucketLimiter
            (remoteIpAddress!, _ =>
                new TokenBucketRateLimiterOptions
                {
                    TokenLimit = myOptions.TokenLimit2,
                    QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                    QueueLimit = myOptions.QueueLimit,
                    ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod),
                    TokensPerPeriod = myOptions.TokensPerPeriod,
                    AutoReplenishment = myOptions.AutoReplenishment
                });
    }

    return RateLimitPartition.GetNoLimiter(IPAddress.Loopback);
});

});

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

```

```

app.UseRouting();
app.UseRateLimiter();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages().RequireRateLimiting(userPolicyName);
app.MapDefaultControllerRoute();

static string GetUserEndPoint(HttpContext context) =>
    $"User {context.User.Identity?.Name ?? "Anonymous"} endpoint:
{context.Request.Path}"
    + $" {context.Connection.RemoteIpAddress}";
static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/a", (HttpContext context) => $"{GetUserEndPoint(context)}
{GetTicks()}")
    .RequireRateLimiting(userPolicyName);


app.MapGet("/b", (HttpContext context) => $"{GetUserEndPoint(context)}
{GetTicks()}")
    .RequireRateLimiting(helloPolicy);

app.MapGet("/c", (HttpContext context) => $"{GetUserEndPoint(context)}
{GetTicks()}");

app.Run();

```

Warning

Creating partitions on client IP addresses makes the app vulnerable to Denial of Service Attacks which employ IP Source Address Spoofing. For more information, see [BCP 38 RFC 2827 Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing](#) .

See [the samples repository](#) for the complete `Program.cs`  file.

The `SampleRateLimiterPolicy` class

```

C#

using System.Threading.RateLimiting;
using Microsoft.AspNetCore.RateLimiting;
using Microsoft.Extensions.Options;
using WebRateLimitAuth.Models;

namespace WebRateLimitAuth;

public class SampleRateLimiterPolicy : IRateLimiterPolicy<string>

```

```

{
    private Func<OnRejectedContext, CancellationToken, ValueTask>?
_onRejected;
    private readonly MyRateLimitOptions _options;

    public SampleRateLimiterPolicy(ILogger<SampleRateLimiterPolicy> logger,
        IOption<MyRateLimitOptions> options)
    {
        _onRejected = (ctx, token) =>
        {
            ctx.HttpContext.Response.StatusCode =
            StatusCodes.Status429TooManyRequests;
            logger.LogWarning($"Request rejected by
{nameof(SampleRateLimiterPolicy)}");
            return ValueTask.CompletedTask;
        };
        _options = options.Value;
    }

    public Func<OnRejectedContext, CancellationToken, ValueTask>? OnRejected
=> _onRejected;

    public RateLimitPartition<string> GetPartition(HttpContext httpContext)
    {
        return RateLimitPartition.GetSlidingWindowLimiter(string.Empty,
            _ => new SlidingWindowRateLimitOptions
            {
                PermitLimit = _options.PermitLimit,
                QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                QueueLimit = _options.QueueLimit,
                Window = TimeSpan.FromSeconds(_options.Window),
                SegmentsPerWindow = _options.SegmentsPerWindow
            });
    }
}

```

In the preceding code, `OnRejected` uses `OnRejectedContext` to set the response status to [429 Too Many Requests](#). The default rejected status is [503 Service Unavailable](#).

Limiter with authorization

The following sample uses JSON Web Tokens (JWT) and creates a partition with the JWT [access token](#). In a production app, the JWT would typically be provided by a server acting as a Security token service (STS). For local development, the dotnet [user-jwts](#) command line tool can be used to create and manage app-specific local JWTs.

C#

```

using System.Threading.RateLimiting;
using Microsoft.AspNetCore.Authentication;

```

```

using Microsoft.Extensions.Primitives;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorization();
builder.Services.AddAuthentication("Bearer").AddJwtBearer();

var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOptions);
var jwtPolicyName = "jwt";

builder.Services.AddRateLimiter(limiterOptions =>
{
    limiterOptions.RejectionStatusCode =
StatusCodes.Status429TooManyRequests;
    limiterOptions.AddPolicy(policyName: jwtPolicyName, partitioner:
HttpContext =>
    {
        var accessToken =
HttpContext.Features.Get<IAuthenticateResultFeature>()?

.AuthenticateResult?.Properties?.GetTokenValue("access_token")?.ToString()
        ?? string.Empty;

        if (!StringValues.IsNullOrEmpty(accessToken))
        {
            return RateLimitPartition.GetTokenBucketLimiter(accessToken, _
=>
                new TokenBucketRateLimiterOptions
                {
                    TokenLimit = myOptions.TokenLimit2,
                    QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                    QueueLimit = myOptions.QueueLimit,
                    ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod),
                    TokensPerPeriod = myOptions.TokensPerPeriod,
                    AutoReplenishment = myOptions.AutoReplenishment
                });
        }

        return RateLimitPartition.GetTokenBucketLimiter("Anon", _ =>
            new TokenBucketRateLimiterOptions
            {
                TokenLimit = myOptions.TokenLimit,
                QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                QueueLimit = myOptions.QueueLimit,
                ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod),
                TokensPerPeriod = myOptions.TokensPerPeriod,
                AutoReplenishment = true
            });
    });
});

```



```

var app = builder.Build();

app.UseAuthorization();
app.UseRateLimiter();

app.MapGet("/", () => "Hello, World!");

app.MapGet("/jwt", (HttpContext context) => $"Hello
{GetUserEndPointMethod(context)}")
    .RequireRateLimiting(jwtPolicyName)
    .RequireAuthorization();

app.MapPost("/post", (HttpContext context) => $"Hello
{GetUserEndPointMethod(context)}")
    .RequireRateLimiting(jwtPolicyName)
    .RequireAuthorization();

app.Run();

static string GetUserEndPointMethod(HttpContext context) =>
    $"Hello {context.User.Identity?.Name ?? "Anonymous"} " +
    $"Endpoint:{context.Request.Path} Method: {context.Request.Method}";

```

Limiter with `ConcurrencyLimiter`, `TokenBucketRateLimiter`, and authorization

The following sample:

- Adds a `ConcurrencyLimiter` with a policy name of "get" that is used on the Razor Pages.
- Adds a `TokenBucketRateLimiter` with a partition for each authorized user and a partition for all anonymous users.
- Sets `RateLimiterOptions.RejectionStatusCode` to [429 Too Many Requests](#).

C#

```

var getPolicyName = "get";
var postPolicyName = "post";
var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOptions);

builder.Services.AddRateLimiter(_ => _
    .AddConcurrencyLimiter(policyName: getPolicyName, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }

```

```

    })
    .AddPolicy(policyName: postPolicyName, partitioner: httpContext =>
    {
        string userName = httpContext.User.Identity?.Name ?? string.Empty;

        if (!StringValues.IsNullOrEmpty(userName))
        {
            return RateLimitPartition.GetTokenBucketLimiter(userName, _ =>
                new TokenBucketRateLimiterOptions
                {
                    TokenLimit = myOptions.TokenLimit2,
                    QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                    QueueLimit = myOptions.QueueLimit,
                    ReplenishmentPeriod =
                        TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod),
                    TokensPerPeriod = myOptions.TokensPerPeriod,
                    AutoReplenishment = myOptions.AutoReplenishment
                });
        }

        return RateLimitPartition.GetTokenBucketLimiter("Anon", _ =>
            new TokenBucketRateLimiterOptions
            {
                TokenLimit = myOptions.TokenLimit,
                QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                QueueLimit = myOptions.QueueLimit,
                ReplenishmentPeriod =
                    TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod),
                TokensPerPeriod = myOptions.TokensPerPeriod,
                AutoReplenishment = true
            });
    });
}

```


See [the samples repository for the complete Program.cs](#) file.

Testing endpoints with rate limiting

Before deploying an app using rate limiting to production, stress test the app to validate the rate limiters and options used. For example, create a [JMeter script](#) with a tool like [BlazeMeter](#) or [Apache JMeter HTTP\(S\) Test Script Recorder](#) and load the script to [Azure Load Testing](#).

Creating partitions with user input makes the app vulnerable to [Denial of Service](#) (DoS) Attacks. For example, creating partitions on client IP addresses makes the app vulnerable to Denial of Service Attacks that employ IP Source Address Spoofing. For more information, see [BCP 38 RFC 2827 Network Ingress Filtering: Defeating Denial of Service Attacks that employ IP Source Address Spoofing](#).

Additional resources

- [Rate limiting middleware](#)  by Maarten Balliauw provides an excellent introduction and overview to rate limiting.
- [Rate limit an HTTP handler in .NET](#)

Request timeouts middleware in ASP.NET Core

Article • 04/10/2024

By [Tom Dykstra](#) 

Apps can apply timeout limits selectively to requests. ASP.NET Core servers don't do this by default since request processing times vary widely by scenario. For example, WebSockets, static files, and calling expensive APIs would each require a different timeout limit. So ASP.NET Core provides middleware that configures timeouts per endpoint as well as a global timeout.

When a timeout limit is hit, a [CancellationToken](#) in [HttpContext.RequestAborted](#) has [IsCancellationRequested](#) set to `true`. [Abort\(\)](#) isn't automatically called on the request, so the application may still produce a success or failure response. The default behavior if the app doesn't handle the exception and produce a response is to return status code 504.

This article explains how to configure the timeout middleware. The timeout middleware can be used in all types of ASP.NET Core apps: Minimal API, Web API with controllers, MVC, and Razor Pages. The sample app is a Minimal API, but every timeout feature it illustrates is also supported in the other app types.

Request timeouts are in the [Microsoft.AspNetCore.Http.Timeouts](#) namespace.

Note: When an app is running in debug mode, the timeout middleware doesn't trigger. This behavior is the same as for [Kestrel timeouts](#). To test timeouts, run the app without the debugger attached.

Add the middleware to the app

Add the request timeouts middleware to the service collection by calling [AddRequestTimeouts](#).

Add the middleware to the request processing pipeline by calling [UseRequestTimeouts](#).

Note

- In apps that explicitly call [UseRouting](#), `UseRequestTimeouts` must be called after `UseRouting`.

Adding the middleware to the app doesn't automatically start triggering timeouts. Timeout limits have to be explicitly configured.

Configure one endpoint or page

For minimal API apps, configure an endpoint to timeout by calling [WithRequestTimeout](#), or by applying the [\[RequestTimeout\]](#) attribute, as shown in the following example:

C#

```
using Microsoft.AspNetCore.Http.Timeouts;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddRequestTimeouts();

var app = builder.Build();
app.UseRequestTimeouts();

app.MapGet("/", async (HttpContext context) => {
    try
    {
        await Task.Delay(TimeSpan.FromSeconds(10), context.RequestAborted);
    }
    catch (TaskCanceledException)
    {
        return Results.Content("Timeout!", "text/plain");
    }

    return Results.Content("No timeout!", "text/plain");
}).WithRequestTimeout(TimeSpan.FromSeconds(2));
// Returns "Timeout!"

app.MapGet("/attribute",
    [RequestTimeout(milliseconds: 2000)] async (HttpContext context) => {
        try
        {
            await Task.Delay(TimeSpan.FromSeconds(10),
context.RequestAborted);
        }
        catch (TaskCanceledException)
        {
            return Results.Content("Timeout!", "text/plain");
        }

        return Results.Content("No timeout!", "text/plain");
    });
// Returns "Timeout!"

app.Run();
```

For apps with controllers, apply the `[RequestTimeout]` attribute to the action method or the controller class. For Razor Pages apps, apply the attribute to the Razor page class.

Configure multiple endpoints or pages

Create named *policies* to specify timeout configuration that applies to multiple endpoints. Add a policy by calling [AddPolicy](#):

```
C#

builder.Services.AddRequestTimeouts(options => {
    options.DefaultPolicy =
        new RequestTimeoutPolicy { Timeout = TimeSpan.FromMilliseconds(1500)
    };
    options.AddPolicy("MyPolicy", TimeSpan.FromSeconds(2));
});
```

A timeout can be specified for an endpoint by policy name:

```
C#

app.MapGet("/namedpolicy", async (HttpContext context) => {
    try
    {
        await Task.Delay(TimeSpan.FromSeconds(10), context.RequestAborted);
    }
    catch (TaskCanceledException)
    {
        return Results.Content("Timeout!", "text/plain");
    }

    return Results.Content("No timeout!", "text/plain");
}).WithRequestTimeout("MyPolicy");
// Returns "Timeout!"
```

The `[RequestTimeout]` attribute can also be used to specify a named policy.

Set global default timeout policy

Specify a policy for the global default timeout configuration:

```
C#

builder.Services.AddRequestTimeouts(options => {
    options.DefaultPolicy =
        new RequestTimeoutPolicy { Timeout = TimeSpan.FromMilliseconds(1500)
    };
});
```

```
options.AddPolicy("MyPolicy", TimeSpan.FromSeconds(2));
});
```

The default timeout applies to endpoints that don't have a timeout specified. The following endpoint code checks for a timeout although it doesn't call the extension method or apply the attribute. The global timeout configuration applies, so the code checks for a timeout:

C#

```
app.MapGet("/", async (HttpContext context) => {
    try
    {
        await Task.Delay(TimeSpan.FromSeconds(10), context.RequestAborted);
    }
    catch
    {
        return Results.Content("Timeout!", "text/plain");
    }

    return Results.Content("No timeout!", "text/plain");
});
// Returns "Timeout!" due to default policy.
```

Specify the status code in a policy

The [RequestTimeoutPolicy](#) class has a property that can automatically set the status code when a timeout is triggered.

C#

```
builder.Services.AddRequestTimeouts(options => {
    options.DefaultPolicy = new RequestTimeoutPolicy {
        Timeout = TimeSpan.FromMilliseconds(1000),
        TimeoutStatusCode = 503
    };
    options.AddPolicy("MyPolicy2", new RequestTimeoutPolicy {
        Timeout = TimeSpan.FromMilliseconds(1000),
        WriteTimeoutResponse = async (HttpContext context) => {
            context.Response.ContentType = "text/plain";
            await context.Response.WriteAsync("Timeout from MyPolicy2!");
        }
    });
});
```

C#

```

app.MapGet("/", async (HttpContext context) => {
    try
    {
        await Task.Delay(TimeSpan.FromSeconds(10), context.RequestAborted);
    }
    catch (TaskCanceledException)
    {
        throw;
    }

    return Results.Content("No timeout!", "text/plain");
});
// Returns status code 503 due to default policy.

```

Use a delegate in a policy

The `RequestTimeoutPolicy` class has a `WriteTimeoutResponse` property that can be used to customize the response when a timeout is triggered.

C#

```

builder.Services.AddRequestTimeouts(options => {
    options.DefaultPolicy = new RequestTimeoutPolicy {
        Timeout = TimeSpan.FromMilliseconds(1000),
        TimeoutStatusCode = 503
    };
    options.AddPolicy("MyPolicy2", new RequestTimeoutPolicy {
        Timeout = TimeSpan.FromMilliseconds(1000),
        WriteTimeoutResponse = async (HttpContext context) => {
            context.Response.ContentType = "text/plain";
            await context.Response.WriteAsync("Timeout from MyPolicy2!");
        }
    });
});

```

C#

```

app.MapGet("/usepolicy2", async (HttpContext context) => {
    try
    {
        await Task.Delay(TimeSpan.FromSeconds(10), context.RequestAborted);
    }
    catch (TaskCanceledException)
    {
        throw;
    }

    return Results.Content("No timeout!", "text/plain");
}).WithRequestTimeout("MyPolicy2");

```



```
// Returns "Timeout from MyPolicy2!" due to WriteTimeoutResponse in
MyPolicy2.
```

Disable timeouts

To disable all timeouts including the default global timeout, use the [\[DisableRequestTimeout\]](#) attribute or the [DisableRequestTimeout](#) extension method:

C#

```
app.MapGet("/disablebyattr", [DisableRequestTimeout] async (HttpContext
context) => {
    try
    {
        await Task.Delay(TimeSpan.FromSeconds(10), context.RequestAborted);
    }
    catch
    {
        return Results.Content("Timeout!", "text/plain");
    }

    return Results.Content("No timeout!", "text/plain");
});
// Returns "No timeout!", ignores default timeout.
```

C#

```
app.MapGet("/disablebyext", async (HttpContext context) => {
    try
    {
        await Task.Delay(TimeSpan.FromSeconds(10), context.RequestAborted);
    }
    catch
    {
        return Results.Content("Timeout!", "text/plain");
    }

    return Results.Content("No timeout!", "text/plain");
}).DisableRequestTimeout();
// Returns "No timeout!", ignores default timeout.
```

Cancel a timeout

To cancel a timeout that has already been started, use the [DisableTimeout\(\)](#) method on [IHttpRequestTimeoutFeature](#). Timeouts cannot be canceled after they've expired.

C#

```
app.MapGet("/canceltimeout", async (HttpContext context) => {  
    var timeoutFeature = context.Features.Get<IHttpRequestTimeoutFeature>();  
    timeoutFeature?.DisableTimeout();  
  
    try  
    {  
        await Task.Delay(TimeSpan.FromSeconds(10), context.RequestAborted);  
    }  
    catch (TaskCanceledException)  
    {  
        return Results.Content("Timeout!", "text/plain");  
    }  
  
    return Results.Content("No timeout!", "text/plain");  
}).WithRequestTimeout(TimeSpan.FromSeconds(1));  
// Returns "No timeout!" since the default timeout is not triggered.
```

See also

- [Microsoft.AspNetCore.Http.Timeouts](#)
- [ASP.NET Core Middleware](#)

Memory management and garbage collection (GC) in ASP.NET Core

Article • 06/17/2024

By [Sébastien Ros](#) and [Rick Anderson](#)

Memory management is complex, even in a managed framework like .NET. Analyzing and understanding memory issues can be challenging. This article:

- Was motivated by many *memory leak* and *GC not working* issues. Most of these issues were caused by not understanding how memory consumption works in .NET Core, or not understanding how it's measured.
- Demonstrates problematic memory use, and suggests alternative approaches.

How garbage collection (GC) works in .NET Core

The GC allocates heap segments where each segment is a contiguous range of memory. Objects placed in the heap are categorized into one of 3 generations: 0, 1, or 2. The generation determines the frequency the GC attempts to release memory on managed objects that are no longer referenced by the app. Lower numbered generations are GC'd more frequently.

Objects are moved from one generation to another based on their lifetime. As objects live longer, they are moved into a higher generation. As mentioned previously, higher generations are GC'd less often. Short term lived objects always remain in generation 0. For example, objects that are referenced during the life of a web request are short lived. Application level [singletons](#) generally migrate to generation 2.

When an ASP.NET Core app starts, the GC:

- Reserves some memory for the initial heap segments.
- Commits a small portion of memory when the runtime is loaded.

The preceding memory allocations are done for performance reasons. The performance benefit comes from heap segments in contiguous memory.

GC.Collect caveats

In general, ASP.NET Core apps in production should **not** use [GC.Collect](#) explicitly. Inducing garbage collections at sub-optimal times can decrease performance significantly.

GC.Collect is useful when investigating memory leaks. Calling `GC.Collect()` triggers a blocking garbage collection cycle that tries to reclaim all objects inaccessible from managed code. It's a useful way to understand the size of the reachable live objects in the heap, and track growth of memory size over time.

Analyzing the memory usage of an app

Dedicated tools can help analyzing memory usage:

- Counting object references
- Measuring how much impact the GC has on CPU usage
- Measuring memory space used for each generation

Use the following tools to analyze memory usage:

- [dotnet-trace](#): Can be used on production machines.
- [Analyze memory usage without the Visual Studio debugger](#)
- [Profile memory usage in Visual Studio](#)

Detecting memory issues

Task Manager can be used to get an idea of how much memory an ASP.NET app is using. The Task Manager memory value:

- Represents the amount of memory that is used by the ASP.NET process.
- Includes the app's living objects and other memory consumers such as native memory usage.

If the Task Manager memory value increases indefinitely and never flattens out, the app has a memory leak. The following sections demonstrate and explain several memory usage patterns.

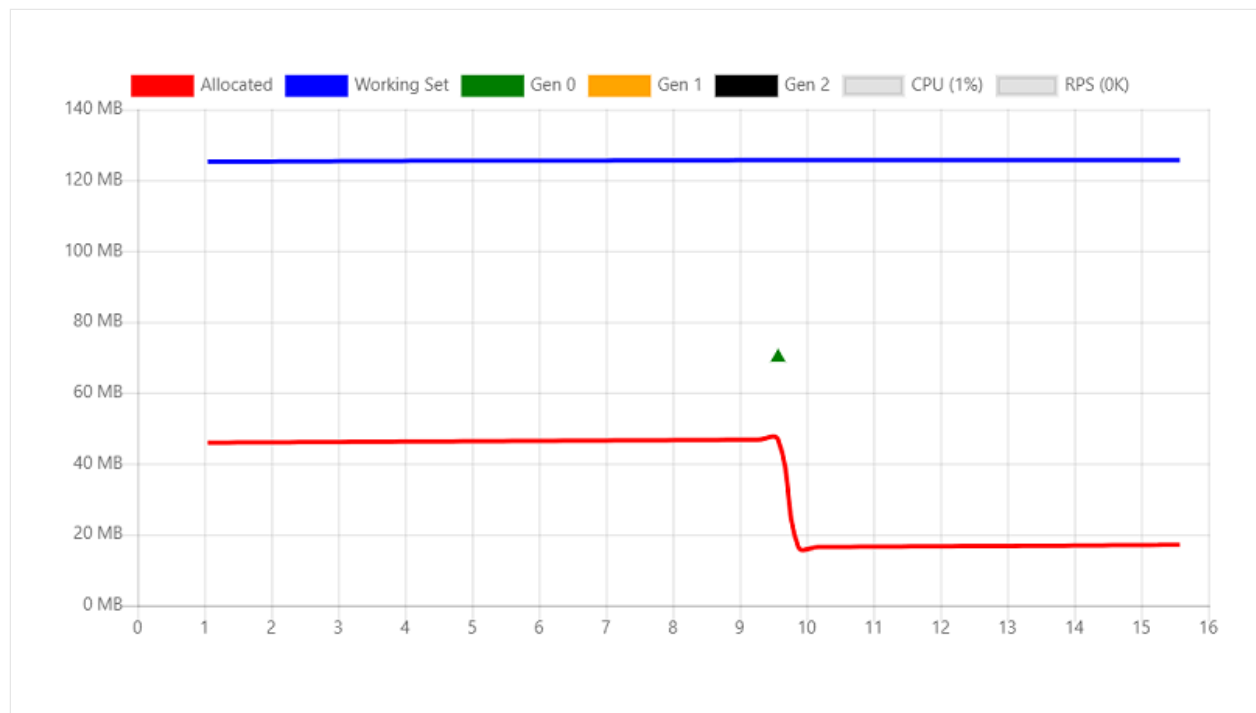
Sample display memory usage app

The [MemoryLeak sample app](#) [↗](#) is available on GitHub. The MemoryLeak app:

- Includes a diagnostic controller that gathers real-time memory and GC data for the app.

- Has an Index page that displays the memory and GC data. The Index page is refreshed every second.
- Contains an API controller that provides various memory load patterns.
- Is not a supported tool, however, it can be used to display memory usage patterns of ASP.NET Core apps.

Run MemoryLeak. Allocated memory slowly increases until a GC occurs. Memory increases because the tool allocates custom object to capture data. The following image shows the MemoryLeak Index page when a Gen 0 GC occurs. The chart shows 0 RPS (Requests per second) because no API endpoints from the API controller have been called.



The chart displays two values for the memory usage:

- **Allocated:** the amount of memory occupied by managed objects
- **Working set:** The set of pages in the virtual address space of the process that are currently resident in physical memory. The working set shown is the same value Task Manager displays.

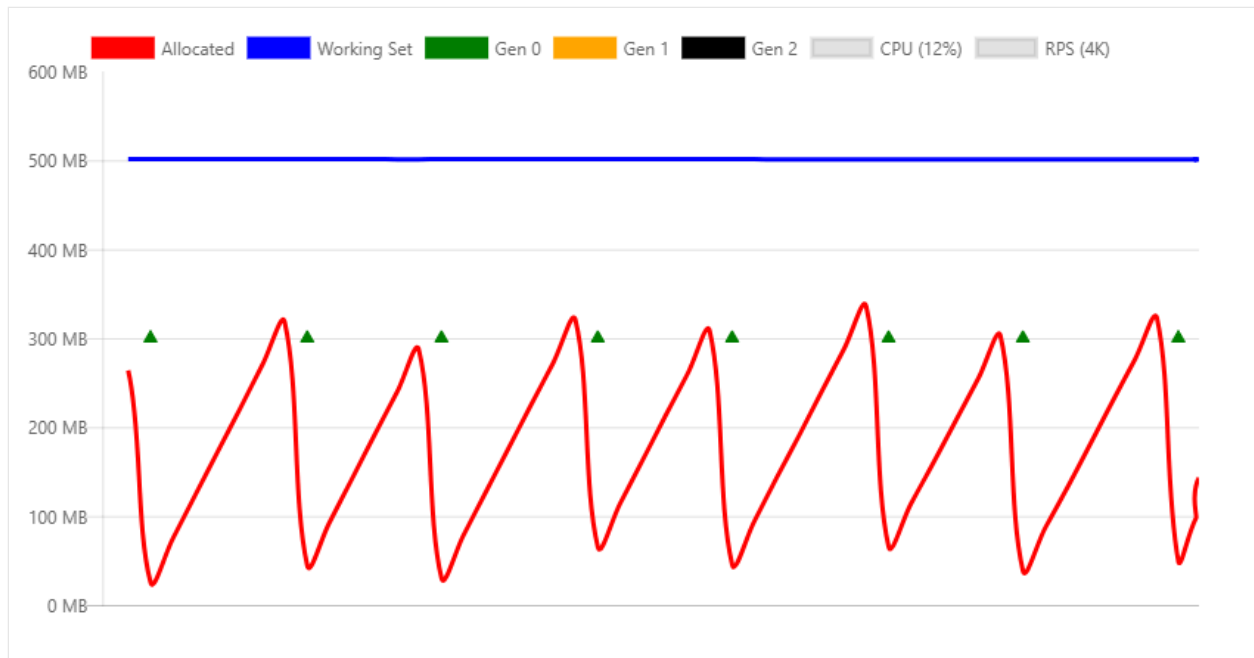
Transient objects

The following API creates a 20-KB String instance and returns it to the client. On each request, a new object is allocated in memory and written to the response. Strings are stored as UTF-16 characters in .NET so each character takes 2 bytes in memory.

C#

```
[HttpGet("bigstring")]  
public ActionResult<string> GetBigString()  
{  
    return new String('x', 10 * 1024);  
}
```

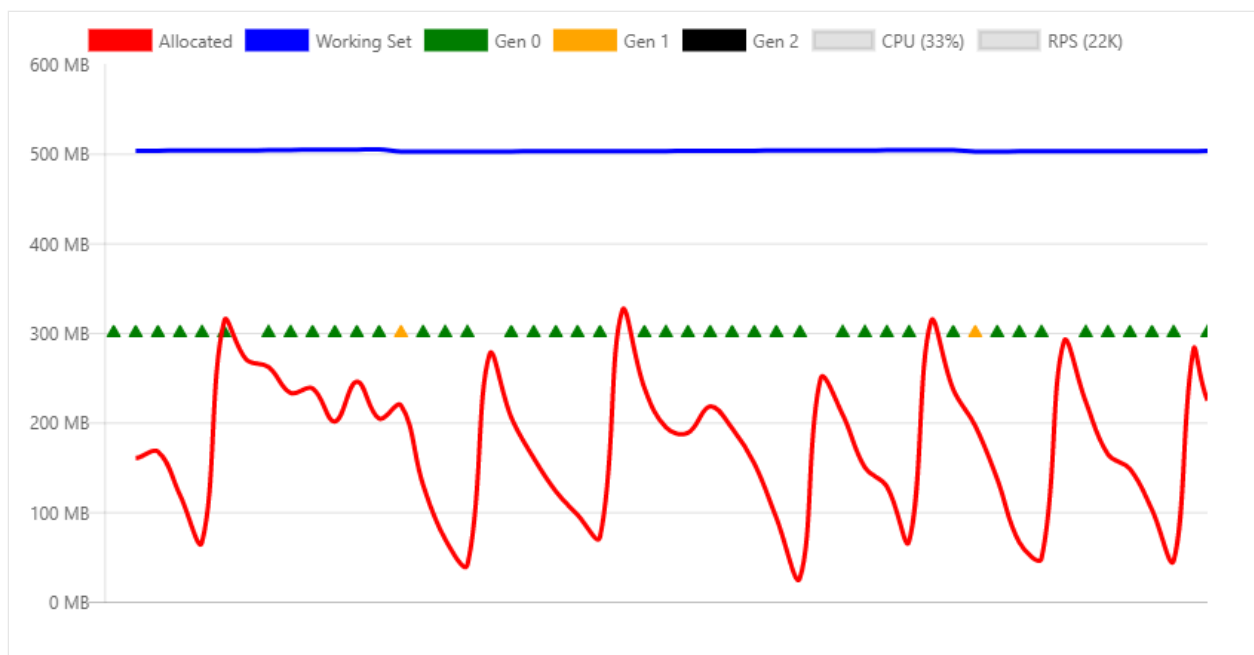
The following graph is generated with a relatively small load in to show how memory allocations are impacted by the GC.



The preceding chart shows:

- 4K RPS (Requests per second).
- Generation 0 GC collections occur about every two seconds.
- The working set is constant at approximately 500 MB.
- CPU is 12%.
- The memory consumption and release (through GC) is stable.

The following chart is taken at the max throughput that can be handled by the machine.



The preceding chart shows:

- 22K RPS
- Generation 0 GC collections occur several times per second.
- Generation 1 collections are triggered because the app allocated significantly more memory per second.
- The working set is constant at approximately 500 MB.
- CPU is 33%.
- The memory consumption and release (through GC) is stable.
- The CPU (33%) is not over-utilized, therefore the garbage collection can keep up with a high number of allocations.

Workstation GC vs. Server GC

The .NET Garbage Collector has two different modes:

- **Workstation GC:** Optimized for the desktop.
- **Server GC.** The default GC for ASP.NET Core apps. Optimized for the server.

The GC mode can be set explicitly in the project file or in the `runtimeconfig.json` file of the published app. The following markup shows setting `ServerGarbageCollection` in the project file:

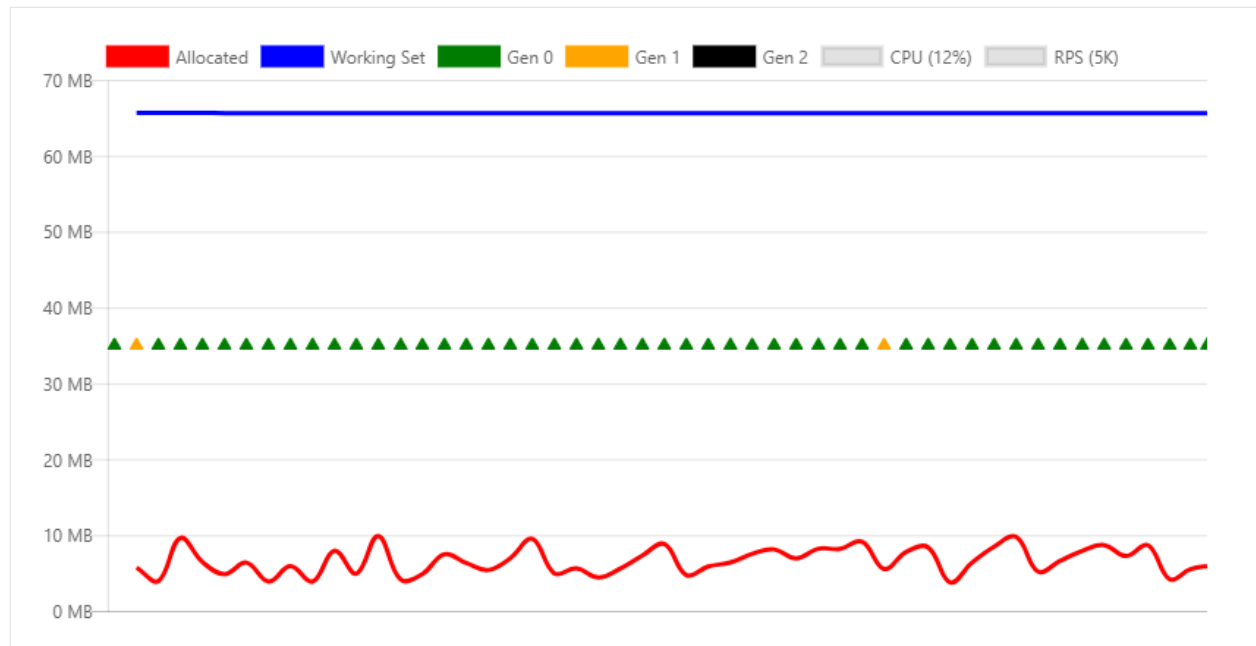
XML

```
<PropertyGroup>
  <ServerGarbageCollection>true</ServerGarbageCollection>
</PropertyGroup>
```

Changing `ServerGarbageCollection` in the project file requires the app to be rebuilt.

Note: Server garbage collection is **not** available on machines with a single core. For more information, see [IsServerGC](#).

The following image shows the memory profile under a 5K RPS using the Workstation GC.



The differences between this chart and the server version are significant:

- The working set drops from 500 MB to 70 MB.
- The GC does generation 0 collections multiple times per second instead of every two seconds.
- GC drops from 300 MB to 10 MB.

On a typical web server environment, CPU usage is more important than memory, therefore the Server GC is better. If memory utilization is high and CPU usage is relatively low, the Workstation GC might be more performant. For example, high density hosting several web apps where memory is scarce.

GC using Docker and small containers

When multiple containerized apps are running on one machine, Workstation GC might be more performant than Server GC. For more information, see [Running with Server GC in a Small Container](#) and [Running with Server GC in a Small Container Scenario Part 1 – Hard Limit for the GC Heap](#).

Persistent object references

The GC cannot free objects that are referenced. Objects that are referenced but no longer needed result in a memory leak. If the app frequently allocates objects and fails to free them after they are no longer needed, memory usage will increase over time.

The following API creates a 20-KB String instance and returns it to the client. The difference with the previous example is that this instance is referenced by a static member, which means it's never available for collection.

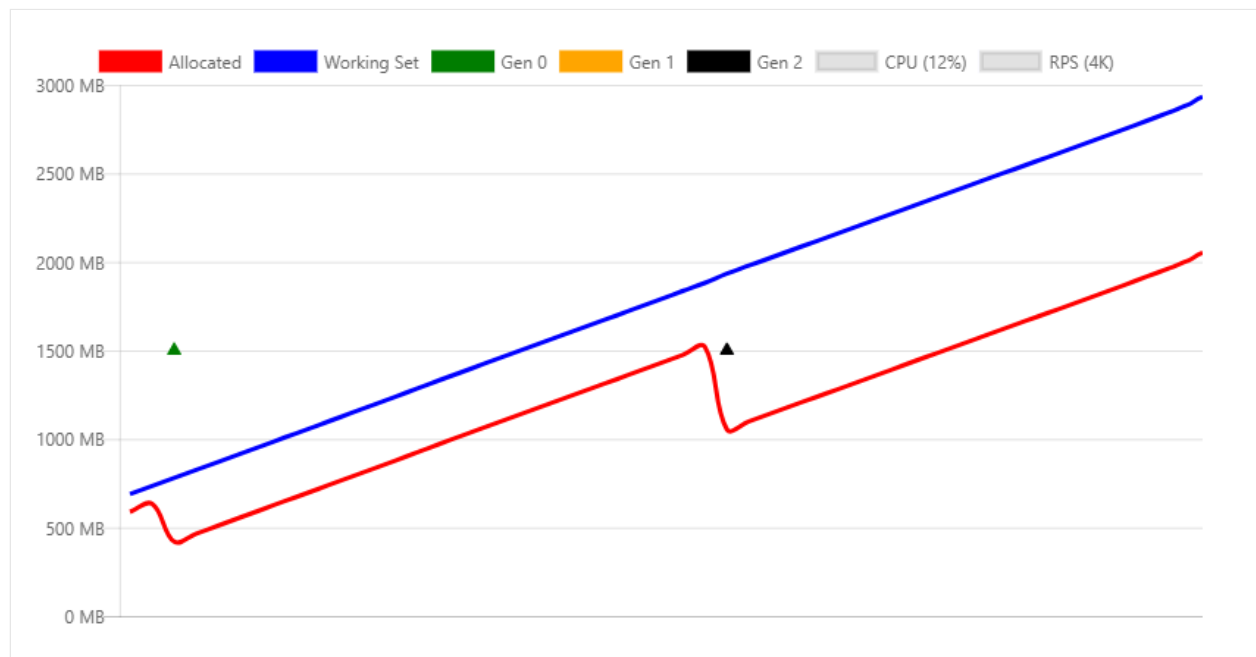
C#

```
private static ConcurrentBag<string> _staticStrings = new
ConcurrentBag<string>();

[HttpGet("staticstring")]
public ActionResult<string> GetStaticString()
{
    var bigString = new String('x', 10 * 1024);
    _staticStrings.Add(bigString);
    return bigString;
}
```

The preceding code:

- Is an example of a typical memory leak.
- With frequent calls, causes app memory to increase until the process crashes with an `OutOfMemory` exception.



In the preceding image:

- Load testing the `/api/staticstring` endpoint causes a linear increase in memory.

- The GC tries to free memory as the memory pressure grows, by calling a generation 2 collection.
- The GC cannot free the leaked memory. Allocated and working set increase with time.

Some scenarios, such as caching, require object references to be held until memory pressure forces them to be released. The [WeakReference](#) class can be used for this type of caching code. A `WeakReference` object is collected under memory pressures. The default implementation of [IMemoryCache](#) uses `WeakReference`.

Native memory

Some .NET Core objects rely on native memory. Native memory can **not** be collected by the GC. The .NET object using native memory must free it using native code.

.NET provides the [IDisposable](#) interface to let developers release native memory. Even if [Dispose](#) is not called, correctly implemented classes call `Dispose` when the [finalizer](#) runs.

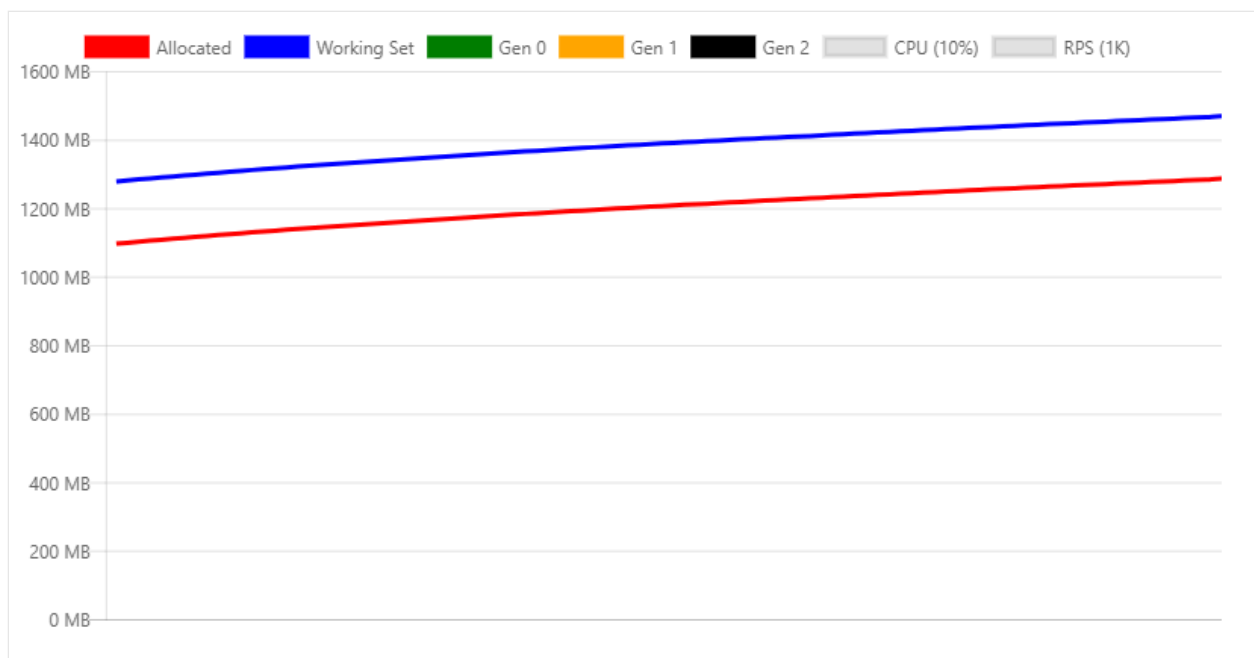
Consider the following code:

C#

```
[HttpGet("fileprovider")]
public void GetFileProvider()
{
    var fp = new PhysicalFileProvider(TempPath);
    fp.Watch("*.*.");
}
```

[PhysicalFileProvider](#) is a managed class, so any instance will be collected at the end of the request.

The following image shows the memory profile while invoking the `fileprovider` API continuously.



The preceding chart shows an obvious issue with the implementation of this class, as it keeps increasing memory usage. This is a known problem that is being tracked in [this issue](#).

The same leak could happen in user code, by one of the following:

- Not releasing the class correctly.
- Forgetting to invoke the `Dispose` method of the dependent objects that should be disposed.

Large object heap

Frequent memory allocation/free cycles can fragment memory, especially when allocating large chunks of memory. Objects are allocated in contiguous blocks of memory. To mitigate fragmentation, when the GC frees memory, it tries to defragment it. This process is called **compaction**. Compaction involves moving objects. Moving large objects imposes a performance penalty. For this reason the GC creates a special memory zone for *large* objects, called the **large object heap** (LOH). Objects that are greater than 85,000 bytes (approximately 83 KB) are:

- Placed on the LOH.
- Not compacted.
- Collected during generation 2 GCs.

When the LOH is full, the GC will trigger a generation 2 collection. Generation 2 collections:

- Are inherently slow.
- Additionally incur the cost of triggering a collection on all other generations.

The following code compacts the LOH immediately:

```
C#

GCSettings.LargeObjectHeapCompactionMode =
    GCLargeObjectHeapCompactionMode.CompactOnce;
GC.Collect();
```

See [LargeObjectHeapCompactionMode](#) for information on compacting the LOH.

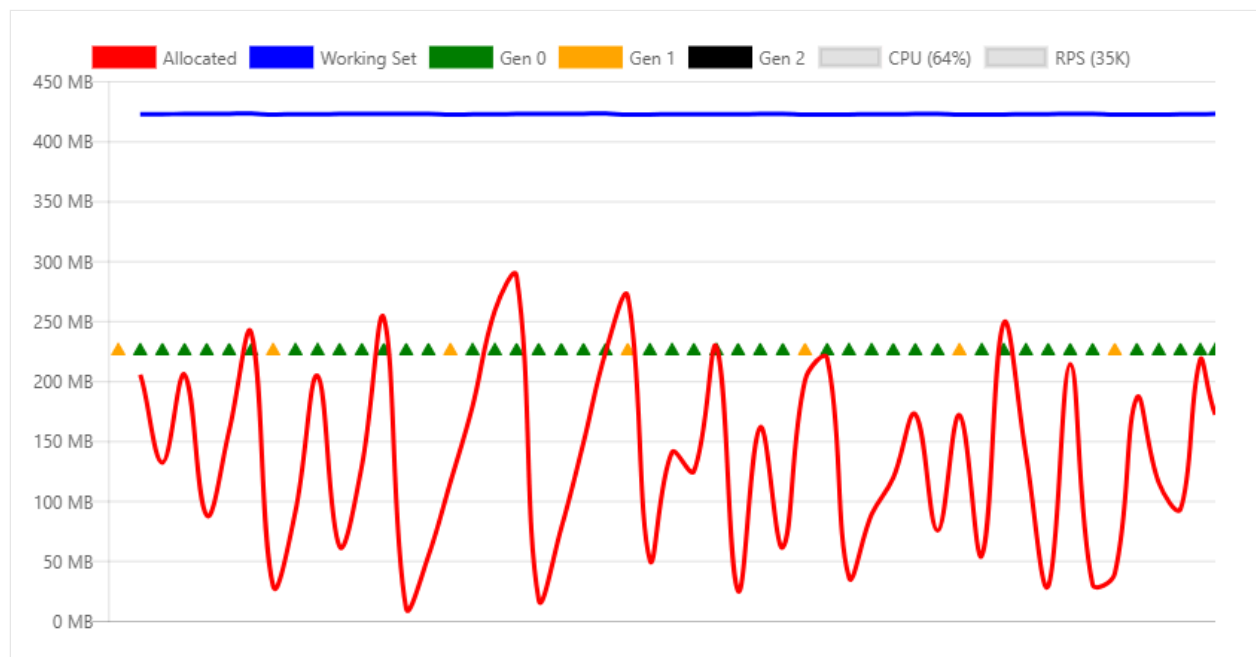
In containers using .NET Core 3.0 and later, the LOH is automatically compacted.

The following API that illustrates this behavior:

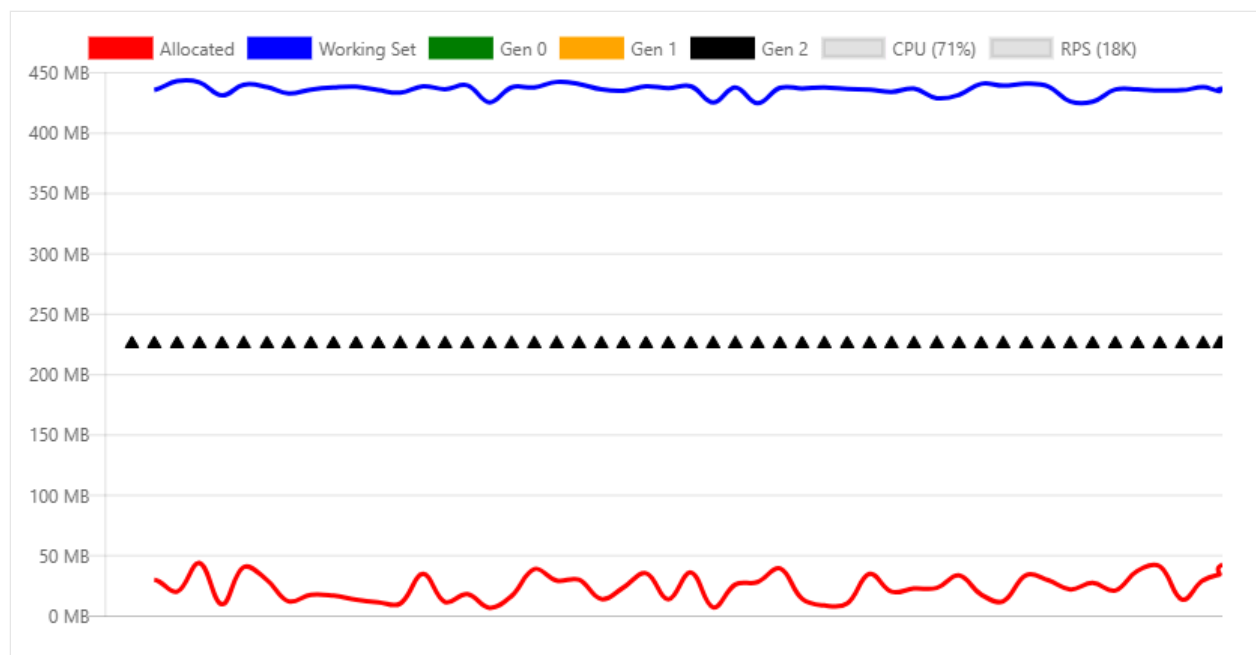
```
C#

[HttpGet("loh/{size=85000}")]
public int GetLOH1(int size)
{
    return new byte[size].Length;
}
```

The following chart shows the memory profile of calling the `/api/loh/84975` endpoint, under maximum load:



The following chart shows the memory profile of calling the `/api/loh/84976` endpoint, allocating *just one more byte*:



Note: The `byte[]` structure has overhead bytes. That's why 84,976 bytes triggers the 85,000 limit.

Comparing the two preceding charts:

- The working set is similar for both scenarios, about 450 MB.
- The under LOH requests (84,975 bytes) shows mostly generation 0 collections.
- The over LOH requests generate constant generation 2 collections. Generation 2 collections are expensive. More CPU is required and throughput drops almost 50%.

Temporary large objects are particularly problematic because they cause gen2 GCs.

For maximum performance, large object use should be minimized. If possible, split up large objects. For example, [Response Caching](#) middleware in ASP.NET Core split the cache entries into blocks less than 85,000 bytes.

The following links show the ASP.NET Core approach to keeping objects under the LOH limit:

- [ResponseCaching/Streams/StreamUtilities.cs](#) [↗](#)
- [ResponseCaching/MemoryResponseCache.cs](#) [↗](#)

For more information, see:

- [Large Object Heap Uncovered](#) [↗](#)
- [Large object heap](#)

HttpClient

Incorrectly using `HttpClient` can result in a resource leak. System resources, such as database connections, sockets, file handles, etc.:

- Are more scarce than memory.
- Are more problematic when leaked than memory.

Experienced .NET developers know to call `Dispose` on objects that implement `IDisposable`. Not disposing objects that implement `IDisposable` typically results in leaked memory or leaked system resources.

`HttpClient` implements `IDisposable`, but should **not** be disposed on every invocation. Rather, `HttpClient` should be reused.

The following endpoint creates and disposes a new `HttpClient` instance on every request:

C#

```
[HttpGet("httpclient1")]
public async Task<int> GetHttpClient1(string url)
{
    using (var httpClient = new HttpClient())
    {
        var result = await httpClient.GetAsync(url);
        return (int)result.StatusCode;
    }
}
```

Under load, the following error messages are logged:

```
fail: Microsoft.AspNetCore.Server.Kestrel[13]
      Connection id "0HLG70PBE1CR1", Request id "0HLG70PBE1CR1:00000031":
      An unhandled exception was thrown by the application.
System.Net.Http.HttpRequestException: Only one usage of each socket address
(protocol/network address/port) is normally permitted --->
System.Net.Sockets.SocketException: Only one usage of each socket
address
(protocol/network address/port) is normally permitted
at System.Net.Http.ConnectHelper.ConnectAsync(String host, Int32 port,
CancellationToken cancellationToken)
```

Even though the `HttpClient` instances are disposed, the actual network connection takes some time to be released by the operating system. By continuously creating new connections, *ports exhaustion* occurs. Each client connection requires its own client port.

One way to prevent port exhaustion is to reuse the same `HttpClient` instance:

C#

```
private static readonly HttpClient _httpClient = new HttpClient();

[HttpGet("httpClient2")]
public async Task<int> GetHttpClient2(string url)
{
    var result = await _httpClient.GetAsync(url);
    return (int)result.StatusCode;
}
```

The `HttpClient` instance is released when the app stops. This example shows that not every disposable resource should be disposed after each use.

See the following for a better way to handle the lifetime of an `HttpClient` instance:

- [HttpClient and lifetime management](#)
- [HTTPClient factory blog](#) ↗

Object pooling

The previous example showed how the `HttpClient` instance can be made static and reused by all requests. Reuse prevents running out of resources.

Object pooling:

- Uses the reuse pattern.
- Is designed for objects that are expensive to create.

A pool is a collection of pre-initialized objects that can be reserved and released across threads. Pools can define allocation rules such as limits, predefined sizes, or growth rate.

The NuGet package [Microsoft.Extensions.ObjectPool](#) ↗ contains classes that help to manage such pools.

The following API endpoint instantiates a `byte` buffer that is filled with random numbers on each request:

C#

```
[HttpGet("array/{size}")]
public byte[] GetArray(int size)
{
    var random = new Random();
    var array = new byte[size];
```

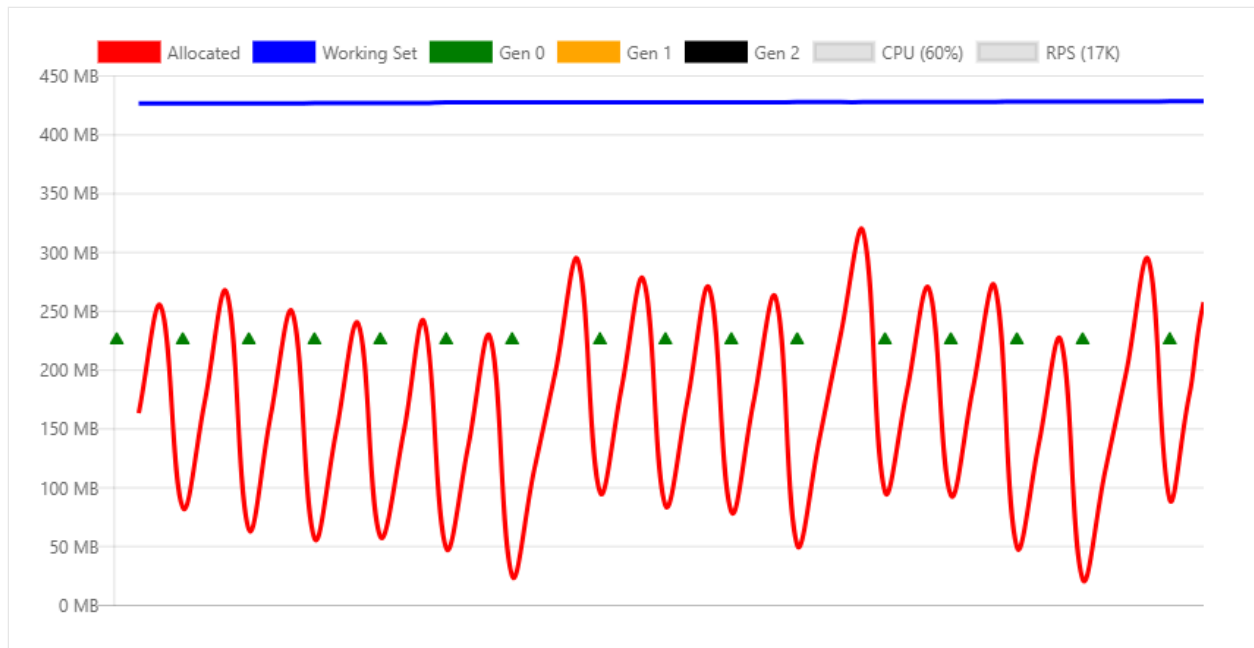
```

        random.NextBytes(array);

        return array;
    }

```

The following chart display calling the preceding API with moderate load:



In the preceding chart, generation 0 collections happen approximately once per second.

The preceding code can be optimized by pooling the `byte` buffer by using `ArrayPool<T>`. A static instance is reused across requests.

What's different with this approach is that a pooled object is returned from the API. That means:

- The object is out of your control as soon as you return from the method.
- You can't release the object.

To set up disposal of the object:

- Encapsulate the pooled array in a disposable object.
- Register the pooled object with `HttpContext.Response.RegisterForDispose`.

`RegisterForDispose` will take care of calling `Dispose` on the target object so that it's only released when the HTTP request is complete.

C#

```

private static ArrayPool<byte> _arrayPool = ArrayPool<byte>.Create();

private class PooledArray : IDisposable
{

```



```

public byte[] Array { get; private set; }

public PooledArray(int size)
{
    Array = _arrayPool.Rent(size);
}

public void Dispose()
{
    _arrayPool.Return(Array);
}
}

[HttpGet("pooledarray/{size}")]
public byte[] GetPooledArray(int size)
{
    var pooledArray = new PooledArray(size);

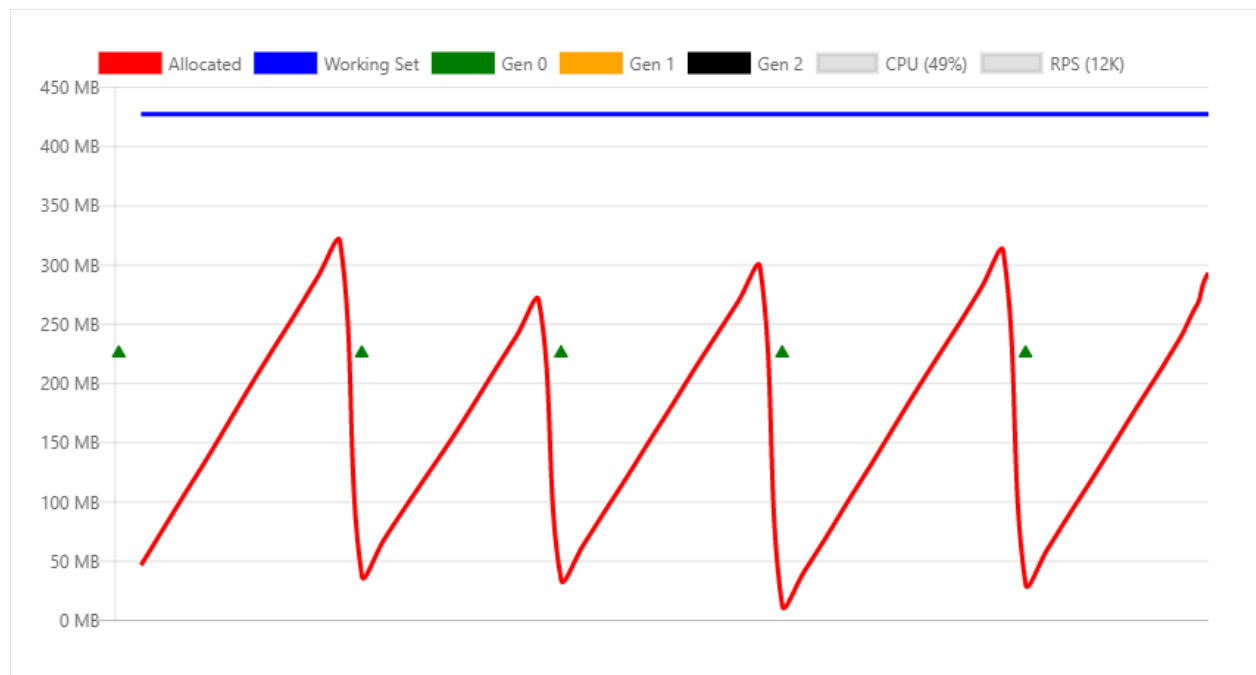
    var random = new Random();
    random.NextBytes(pooledArray.Array);

    HttpContext.Response.RegisterForDispose(pooledArray);

    return pooledArray.Array;
}

```

Applying the same load as the non-pooled version results in the following chart:



The main difference is allocated bytes, and as a consequence much fewer generation 0 collections.

Additional resources

- [Garbage Collection](#)
- [Understanding different GC modes with Concurrency Visualizer](#) ↗
- [Large Object Heap Uncovered](#) ↗
- [Large object heap](#)

Deploying and scaling an ASP.NET Core app on Azure Container Apps

Article • 09/18/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

Apps deployed to Azure that experience intermittent high demand benefit from scalability to meet demand. Scalable apps can scale out to ensure capacity during workload peaks and then scale down automatically when the peak drops, which can lower costs. Horizontal scaling (scaling out) adds new instances of a resource, such as VMs or database replicas. This article demonstrates how to deploy a horizontally scalable ASP.NET Core app to [Azure container apps](#) by completing the following tasks:

1. [Set up the sample project](#)
2. [Deploy the app to Azure Container Apps](#)
3. [Scale and troubleshoot the app](#)
4. [Create the Azure Services](#)
5. [Connect the Azure services](#)
6. [Configure and redeploy the app](#)

This article uses Razor Pages, but most of it applies to other ASP.NET Core apps.

In some cases, basic ASP.NET Core apps are able to scale without special considerations. However, apps that utilize certain framework features or architectural patterns require extra configurations, including the following:

- **Secure form submissions:** Razor Pages, MVC and Web API apps often rely on form submissions. By default these apps use [cross site forgery tokens](#) and internal data protection services to secure requests. When deployed to the cloud, these apps must be configured to manage data protection service concerns in a secure, centralized location.
- **SignalR circuits:** Blazor Server apps require the use of a centralized [Azure SignalR service](#) in order to securely scale. These services also utilize the data protection

services mentioned previously.

- **Centralized caching or state management services:** Scalable apps may use [Azure Cache for Redis](#) to provide distributed caching. [Azure storage](#) may be needed to store state for frameworks such as [Microsoft Orleans](#), which can help write apps that manage state across many different app instances.

The steps in this article demonstrate how to properly address the preceding concerns by deploying a scalable app to Azure Container Apps. Most of the concepts in this tutorial also apply when scaling [Azure App Service](#) instances.

Set up the sample project

Use the GitHub Explorer sample app to follow along with this tutorial. Clone the app from GitHub using the following command:

```
.NET CLI
```

```
git clone "https://github.com/dotnet/AspNetCore.Docs.Samples.git"
```

Navigate to the `/tutorials/scalable-razor-apps/start` folder and open the `ScalableRazor.csproj`.

The sample app uses a search form to browse GitHub repositories by name. The form relies on the built-in ASP.NET Core data protection services to handle antiforgery concerns. By default, when the app scales horizontally on Container Apps, the data protection service throws an exception.

Test the app

1. Launch the app in Visual Studio. The project includes a Docker file, which means that the arrow next to the run button can be selected to start the app using either a Docker Desktop setup or the standard ASP.NET Core local web server.

Use the search form to browse for GitHub repositories by name.

Welcome to the GitHub searcher!

Enter the name of a GitHub organization such as "Microsoft" or "Azure" to browse its repositories.

Name	Description	Link
HealthVault-Mobile-iOS-Library	The HealthVault team has recently added the capability to write applications that will run on Mobile Devices and connect directly to the	<input type="button" value="Browse"/>

Deploy the app to Azure Container Apps

Visual Studio is used to deploy the app to Azure Container Apps. Container apps provide a managed service designed to simplify hosting containerized apps and microservices.

! Note

Many of the resources created for the app require a location. For this app, location isn't important. A real app should select a location closest to the clients. You may want to select a location near you.

1. In Visual Studio solution explorer, right click on the top level project node and select **Publish**.
2. In the publishing dialog, select **Azure** as the deployment target, and then select **Next**.
3. For the specific target, select **Azure Container Apps (Linux)**, and then select **Next**.
4. Create a new container app to deploy to. Select the green + icon to open a new dialog and enter the following values:

Azure Container Apps
Create new

Microsoft

Container app name
razorscaling-app-20220915131547

Subscription name
C&L Cross Service Content Team Testing

Resource group
msdocs-scalable-razor (South Central US) [New...](#)

Container Apps environment
scalablerazorenv (South Central US) [New...](#)

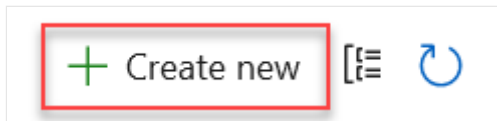
Container name
razorscaling

Export... Create Cancel

- **Container app name:** Leave the default value or enter a name.
- **Subscription name:** Select the subscription to deploy to.
- **Resource group:** Select **New** and create a new resource group called *msdocs-scalable-razor*.
- **Container apps environment:** Select **New** to open the container apps environment dialog and enter the following values:
 - **Environment name:** Keep the default value.
 - **Location:** Select a location near you.
 - **Azure Log Analytics Workspace:** Select **New** to open the log analytics workspace dialog.
 - **Name:** Leave the default value.
 - **Location:** Select a location near you and then select **Ok** to close the dialog.
 - Select **Ok** to close the container apps environment dialog.
- Select **Create** to close the original container apps dialog. Visual Studio creates the container app resource in Azure.

5. Once the resource is created, make sure it's selected in the list of container apps, and then select **Next**.

6. You'll need to create an Azure Container Registry to store the published image artifact for your app. Select the green + icon on the container registry screen.



7. Leave the default values, and then select **Create**.

A screenshot of the 'Create new' dialog box for Azure Container Registry. The dialog has a title bar with the Azure Container Registry logo and the text 'Create new'. Below the title bar is a dropdown menu for 'Microsoft'. The main area contains several input fields: 'DNS prefix' with the value 'RazorScaling20220915131713', 'Subscription name' with the value 'C&L Cross Service Content Team Testing', 'Resource group' with the value 'msdocs-scalable-razor (South Central US)' and a 'New...' link, 'SKU' with the value 'Standard', and 'Registry location' with the value 'South Central US'. At the bottom, there are three buttons: 'Export...', 'Create', and 'Cancel'.

8. After the container registry is created, make sure it's selected, and then select finish to close the dialog workflow and display a summary of the publishing profile.

If Visual Studio prompts you to enable the Admin user to access the published docker container, select **Yes**.

9. Select **Publish** in the upper right of the publishing profile summary to deploy the app to Azure.

When the deployment finishes, Visual Studio launches the browser to display the hosted app. Search for **Microsoft** in the form field, and a list of repositories is displayed.

Scale and troubleshoot the app

The app is currently working without any issues, but we'd like to scale the app across more instances in anticipation of high traffic volumes.

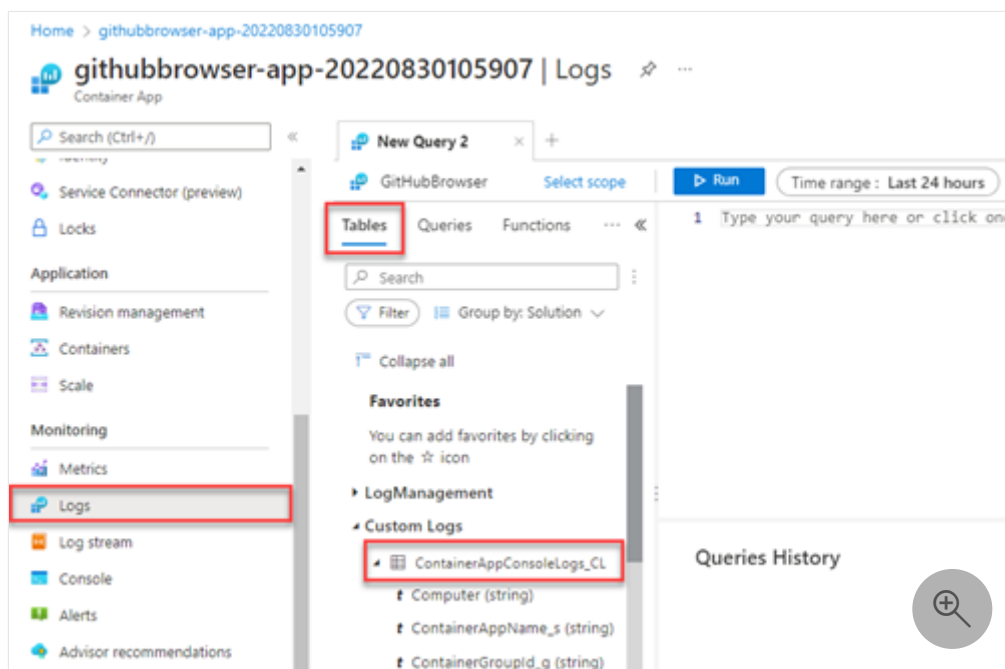
1. In the Azure portal, search for the `razorscaling-app-****` container app in the top level search bar and select it from the results.
2. On the overview page, select **Scale** from the left navigation, and then select **+ Edit and deploy**.
3. On the revisions page, switch to the **Scale** tab.
4. Set both the min and max instances to **4** and then select **Create**. This configuration change guarantees your app is scaled horizontally across four instances.

Navigate back to the app. When the page loads, at first it appears everything is working correctly. However, when a search term is entered and submitted, an error may occur. If an error is not displayed, submit the form several more times.

Troubleshooting the error

It's not immediately apparent why the search requests are failing. The browser tools indicate a 400 Bad Request response was sent back. However, you can use the logging features of container apps to diagnose errors occurring in your environment.

1. On the overview page of the container app, select **Logs** from the left navigation.
2. On the **Logs** page, close the pop-up that opens and navigate to the **Tables** tab.
3. Expand the **Custom Logs** item to reveal the **ContainerAppConsoleLogs_CL** node. This table holds various logs for the container app that can be queried to troubleshoot problems.



4. In the query editor, compose a basic query to search the **ContainerAppConsoleLogs_CL** Logs table for recent exceptions, such as the following script:

KQL

```
ContainerAppConsoleLogs_CL
| where Log_s contains "exception"
| sort by TimeGenerated desc
| limit 500
| project ContainerAppName_s, Log_s
```

The preceding query searches the **ContainerAppConsoleLogs_CL** table for any rows that contain the word exception. The results are ordered by the time generated, limited to 500 results, and only include the **ContainerAppName_s** and **Log_s** columns to make the results easier to read.

5. Select **Run**, a list of results is displayed. Read through the logs and note that most of them are related to antiforgery tokens and cryptography.

The screenshot shows the Azure Data Explorer interface. The query editor on the left contains the following KQL script:

```
1 ContainerAppConsoleLogs_CL
2 | where Log_s contains "exception"
3 | sort by TimeGenerated desc
4 | limit 500
5 | project ContainerAppName_s, Log_s
```

The results pane on the right displays a table with two columns: **ContainerAppName_s** and **Log_s**. The results show several log entries for the **githubbrowser-app-20220830105907** container, all containing exceptions. The exceptions include:

- System.Security.Cryptography.CryptographicException: The key [c4fee72...
- End of inner exception stack trace ...
- An exception was thrown while deserializing the token.
- Microsoft.AspNetCore.Antiforgery.AntiforgeryValidationException: The antifor...
- An unhandled exception has occurred while executing the request.
- System.InvalidOperationException: An invalid request URI was provided. Ethe...
- at Microsoft.AspNetCore.Diagnostics.ExceptionHandlerMiddleware.<Invoke>...
- [41m[30mfail[39m[22m[49m: Microsoft.AspNetCore.Diagnostics.Exception...
- [41m[30mfail[39m[22m[49m: Microsoft.AspNetCore.Diagnostics.Exception...

❗ Important

The errors in the app are caused by the .NET data protection services. When multiple instances of the app are running, there is no guarantee that the HTTP POST request to submit the form is routed to the same container that initially loaded the page from the HTTP GET request. If the requests are handled by different instances, the antiforgery tokens aren't handled correctly and an exception occurs.

In the steps ahead, this issue is resolved by centralizing the data protection keys in an Azure storage service and protecting them with Key Vault.

Create the Azure Services

To resolve the preceding errors, the following services are created and connected to the app:

- **Azure Storage Account:** Handles storing data for the Data Protection Services. Provides a centralized location to store key data as the app scales. Storage accounts can also be used to hold documents, queue data, file shares, and almost any type of blob data.
- **Azure KeyVault:** This service stores secrets for an app, and is used to help manage encryption concerns for the Data Protection Services.

Create the storage account service

1. In the Azure portal search bar, enter **Storage accounts** and select the matching result.
2. On the storage accounts listing page, select + **Create**.
3. On the **Basics** tab, enter the following values:
 - **Subscription:** Select the same subscription that you chose for the container app.
 - **Resource Group:** Select the *msdocs-scalable-razor* resource group you created previously.
 - **Storage account name:** Name the account *scalablerazorstorageXXXX* where the X's are random numbers of your choosing. This name must be unique across all of Azure.
 - **Region:** Select the same region you previously selected.
4. Leave the rest of the values at their default and select **Review**. After Azure validates the inputs, select **Create**.

Azure provisions the new storage account. When the task completes, choose **Go to resource** to view the new service.

Create the storage container

Create a Container to store the app's data protection keys.

1. On the overview page for the new storage account, select **Storage browser** on the left navigation.
2. Select **Blob containers**.
3. Select + **Add container** to open the **New container** flyout menu.

4. Enter a name of *scalablerazorkeys*, leave the rest of the settings at their defaults, and then select **Create**.

The new containers appear on the page list.

Create the key vault service

Create a key vault to hold the keys that protect the data in the blob storage container.

1. In the Azure portal search bar, enter `Key Vault` and select the matching result.
2. On the key vault listing page, select **+ Create**.
3. On the **Basics** tab, enter the following values:
 - **Subscription**: Select the same subscription that was previously selected.
 - **Resource Group**: Select the *msdocs-scalable-razor* resource group previously created.
 - **Key Vault name**: Enter the name *scalablerazervaultXXXX*.
 - **Region**: Select a region near your location.
4. Leave the rest of the settings at their default, and then select **Review + create**.
Wait for Azure to validate your settings, and then select **Create**.

Azure provisions the new key vault. When the task completes, select **Go to resource** to view the new service.

Create the key

Create a secret key to protect the data in the blob storage account.

1. On the main overview page of the key vault, select **Keys** from the left navigation.
2. On the **Create a key** page, select **+ Generate/Import** to open the **Create a key** flyout menu.
3. Enter *razorkey* in the **Name** field. Leave the rest of the settings at their default values and then select **Create**. A new key appears on the key list page.

Connect the Azure Services

The Container App requires a secure connection to the storage account and the key vault services in order to resolve the data protection errors and scale correctly. The new services are connected together using the following steps:

Security role assignments through Service Connector and other tools typically take a minute or two to propagate, and in some rare cases can take up to eight minutes.

Connect the storage account

1. In the Azure portal, navigate to the Container App overview page.
2. On the left navigation, select **Service connector**
3. On the Service Connector page, choose + **Create** to open the **Creation Connection** flyout panel and enter the following values:
 - **Container:** Select the Container App created previously.
 - **Service type:** Choose **Storage - blob**.
 - **Subscription:** Select the subscription previously used.
 - **Connection name:** Leave the default value.
 - **Storage account:** Select the storage account created previously.
 - **Client type:** Select **.NET**.
4. Select **Next: Authentication** to progress to the next step.
5. Select **System assigned managed identity** and choose **Next: Networking**.
6. Leave the default networking options selected, and then select **Review + Create**.
7. After Azure validates the settings, select **Create**.

The service connector enables a system-assigned managed identity on the container app. It also assigns a role of **Storage Blob Data Contributor** to the identity so it can perform data operations on the storage containers.

Connect the key vault

1. In the Azure portal, navigate to your Container App overview page.
2. On the left navigation, select **Service connector**.
3. On the Service Connector page, choose + **Create** to open the **Creation Connection** flyout panel and enter the following values:
 - **Container:** Select the container app created previously.
 - **Service type:** Choose **Key Vault**.
 - **Subscription:** Select the subscription previously used.
 - **Connection name:** Leave the default value.
 - **Key vault:** Select the key vault created previously.
 - **Client type:** Select **.NET**.
4. Select **Next: Authentication** to progress to the next step.
5. Select **System assigned managed identity** and choose **Next: Networking**.

6. Leave the default networking options selected, and then select **Review + Create**.
7. After Azure validates the settings, select **Create**.

The service connector assigns a role to the identity so it can perform data operations on the key vault keys.

Configure and redeploy the app

The necessary Azure resources have been created. In this section the app code is configured to use the new resources.

1. Install the following NuGet packages:

- **Azure.Identity**: Provides classes to work with the Azure identity and access management services.
- **Microsoft.Extensions.Azure**: Provides helpful extension methods to perform core Azure configurations.
- **Azure.Extensions.AspNetCore.DataProtection.Blobs**: Allows storing ASP.NET Core DataProtection keys in Azure Blob Storage so that keys can be shared across several instances of a web app.
- **Azure.Extensions.AspNetCore.DataProtection.Keys**: Enables protecting keys at rest using the Azure Key Vault Key Encryption/Wrapping feature.

.NET CLI

```
dotnet add package Azure.Identity
dotnet add package Microsoft.Extensions.Azure
dotnet add package Azure.Extensions.AspNetCore.DataProtection.Blobs
dotnet add package Azure.Extensions.AspNetCore.DataProtection.Keys
```

2. Update `Program.cs` with the following highlighted code:

C#

```
using Azure.Identity;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.Azure;

var builder = WebApplication.CreateBuilder(args);
var BlobStorageUri = builder.Configuration["AzureURIs:BlobStorage"];
var KeyVaultURI = builder.Configuration["AzureURIs:KeyVault"];

builder.Services.AddRazorPages();
builder.Services.AddHttpClient();
builder.Services.AddServerSideBlazor();
```

```

builder.Services.AddAzureClientsCore();

builder.Services.AddDataProtection()
    .PersistKeysToAzureBlobStorage(new Uri(BlobStorageUri),
                                    new
DefaultAzureCredential())
    .ProtectKeysWithAzureKeyVault(new Uri(KeyVaultURI),
                                   new
DefaultAzureCredential());
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();

```

The preceding changes allow the app to manage data protection using a centralized, scalable architecture. `DefaultAzureCredential` discovers the managed identity configurations enabled earlier when the app is redeployed.

Update the placeholders in `AzureURIs` section of the `appsettings.json` file to include the following:

1. Replace the `<storage-account-name>` placeholder with the name of the `scalablerazorstorageXXXX` storage account.
2. Replace the `<container-name>` placeholder with the name of the `scalablerazorkeys` storage container.
3. Replace the `<key-vault-name>` placeholder with the name of the `scalablerazorbvaultXXXX` key vault.
4. Replace the `<key-name>` placeholder in the key vault URI with the `razorkey` name created previously.

```

{
  "GitHubURL": "https://api.github.com",
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "AzureURIs": {
    "BlobStorage": "https://<storage-account-name>.blob.core.windows.net/<container-name>/keys.xml",
    "KeyVault": "https://<key-vault-name>.vault.azure.net/keys/<key-name>/"
  }
}

```

Redeploy the app

The app is now configured correctly to use the Azure services created previously. Redeploy the app for the code changes to be applied.

1. Right click on the project node in the solution explorer and select **Publish**.
2. On the publishing profile summary view, select the **Publish** button in the upper right corner.

Visual Studio redeploys the app to the container apps environment created previously. When the processes finished, the browser launches to the app home page.

Test the app again by searching for *Microsoft* in the search field. The page should now reload with the correct results every time you submit.

Configure roles for local development

The existing code and configuration of the app can also work while running locally during development. The `DefaultAzureCredential` class configured previously is able to pick up local environment credentials to authenticate to Azure Services. You'll need to assign the same roles to your own account that were assigned to your app's managed identity in order for the authentication to work. This should be the same account you use to log into Visual Studio or the Azure CLI.

Sign-in to your local development environment

You'll need to be signed in to the Azure CLI, Visual Studio, or Azure PowerShell for your credentials to be picked up by `DefaultAzureCredential`.



Assign roles to your developer account

1. In the Azure portal, navigate to the `scalablerazor****` storage account created previously.
2. Select **Access Control (IAM)** from the left navigation.
3. Choose + **Add** and then **Add role assignment** from the drop-down menu.
4. On the **Add role assignment** page, search for `Storage blob data contributor`, select the matching result, and then select **Next**.
5. Make sure **User, group, or service principal** is selected, and then select + **Select members**.
6. In the **Select members** flyout, search for your own `user@domain` account and select it from the results.
7. Choose **Next** and then select **Review + assign**. After Azure validates the settings, select **Review + assign** again.

As previously stated, role assignment permissions might take a minute or two to propagate, or in rare cases up to eight minutes.

Repeat the previous steps to assign a role to your account so that it can access the key vault service and secret.

1. In the Azure portal, navigate to the `razorscalingkeys` key vault created previously.
2. Select **Access Control (IAM)** from the left navigation.
3. Choose + **Add** and then **Add role assignment** from the drop-down menu.
4. On the **Add role assignment** page, search for `Key Vault Crypto Service Encryption User`, select the matching result, and then select **Next**.
5. Make sure **User, group, or service principal** is selected, and then select + **Select members**.
6. In the **Select members** flyout, search for your own `user@domain` account and select it from the results.

7. Choose **Next** and then select **Review + assign**. After Azure validates your settings, select **Review + assign** again.

You may need to wait again for this role assignment to propagate.

You can then return to Visual Studio and run the app locally. The code should continue to function as expected. `DefaultAzureCredential` uses your existing credentials from Visual Studio or the Azure CLI.

Reliable web app patterns

See *The Reliable Web App Pattern for .NET* [YouTube videos](#) and [article](#) for guidance on creating a modern, reliable, performant, testable, cost-efficient, and scalable ASP.NET Core app, whether from scratch or refactoring an existing app.

Object reuse with ObjectPool in ASP.NET Core

Article • 06/17/2024

By [Günther Foidl](#) , [Steve Gordon](#) , and [Samson Amaugo](#) 

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

`Microsoft.Extensions.ObjectPool` is part of the ASP.NET Core infrastructure that supports keeping a group of objects in memory for reuse rather than allowing the objects to be garbage collected. All the static and instance methods in `Microsoft.Extensions.ObjectPool` are thread-safe.

Apps might want to use the object pool if the objects that are being managed are:

- Expensive to allocate/initialize.
- Represent a limited resource.
- Used predictably and frequently.

For example, the ASP.NET Core framework uses the object pool in some places to reuse `StringBuilder` instances. `StringBuilder` allocates and manages its own buffers to hold character data. ASP.NET Core regularly uses `StringBuilder` to implement features, and reusing them provides a performance benefit.

Object pooling doesn't always improve performance:

- Unless the initialization cost of an object is high, it's usually slower to get the object from the pool.
- Objects managed by the pool aren't de-allocated until the pool is de-allocated.

Use object pooling only after collecting performance data using realistic scenarios for your app or library.

NOTE: The `ObjectPool` doesn't place a limit on the number of objects that it allocates, it places a limit on the number of objects it retains.

ObjectPool concepts

When [DefaultObjectPoolProvider](#) is used and `T` implements `IDisposable`:

- Items that are *not* returned to the pool will be disposed.
- When the pool gets disposed by DI, all items in the pool are disposed.

NOTE: After the pool is disposed:

- Calling `Get` throws an `ObjectDisposedException`.
- Calling `Return` disposes the given item.

Important `ObjectPool` types and interfaces:

- [ObjectPool<T>](#) : The basic object pool abstraction. Used to get and return objects.
- [PooledObjectPolicy<T>](#) : Implement this to customize how an object is created and how it's reset when returned to the pool. This can be passed into an object pool that's constructed directly.
- [IResettable](#) : Automatically resets the object when returned to an object pool.

The ObjectPool can be used in an app in multiple ways:

- Instantiating a pool.
- Registering a pool in [Dependency injection](#) (DI) as an instance.
- Registering the `ObjectPoolProvider<>` in DI and using it as a factory.

How to use ObjectPool

Call [Get](#) to get an object and [Return](#) to return the object. There's no requirement to return every object. If an object isn't returned, it will be garbage collected.

ObjectPool sample

The following code:

- Adds `ObjectPoolProvider` to the [Dependency injection](#) (DI) container.
- Implements the `IResettable` interface to automatically clear the contents of the buffer when returned to the object pool.

C#

```
using Microsoft.Extensions.DependencyInjection.Extensions;  
using Microsoft.Extensions.ObjectPool;
```

```

using System.Security.Cryptography;

var builder = WebApplication.CreateBuilder(args);

builder.Services.TryAddSingleton<ObjectPoolProvider,
DefaultObjectPoolProvider>();

builder.Services.TryAddSingleton<ObjectPool<ReusableBuffer>>(serviceProvider
=>
{
    var provider = serviceProvider.GetRequiredService<ObjectPoolProvider>();
    var policy = new DefaultPooledObjectPolicy<ReusableBuffer>();
    return provider.Create(policy);
});

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

// return the SHA256 hash of a word
// https://localhost:7214/hash/SamsonAmaugo
app.MapGet("/hash/{name}", (string name, ObjectPool<ReusableBuffer>
bufferPool) =>
{
    var buffer = bufferPool.Get();
    try
    {
        // Set the buffer data to the ASCII values of a word
        for (var i = 0; i < name.Length; i++)
        {
            buffer.Data[i] = (byte)name[i];
        }

        Span<byte> hash = stackalloc byte[32];
        SHA256.HashData(buffer.Data.AsSpan(0, name.Length), hash);
        return "Hash: " + Convert.ToHexString(hash);
    }
    finally
    {
        // Data is automatically reset because this type implemented
        IResettable
        bufferPool.Return(buffer);
    }
});

app.Run();

public class ReusableBuffer : IResettable
{
    public byte[] Data { get; } = new byte[1024 * 1024]; // 1 MB

    public bool TryReset()
    {
        Array.Clear(Data);
        return true;
    }
}

```

```
}  
}
```

NOTE: When the pooled type `T` doesn't implement `IResettable`, then a custom `PooledObjectPolicy<T>` can be used to reset the state of the objects before they are returned to the pool.

Response compression in ASP.NET Core

Article • 07/01/2024

Network bandwidth is a limited resource. Reducing the size of the response usually increases the responsiveness of an app, often dramatically. One way to reduce payload sizes is to compress an app's responses.

Compression with HTTPS

Compressed responses over secure connections can be controlled with the `EnableForHttps` option, which is disabled by default because of the security risk. Using compression with dynamically generated pages can expose the app to [CRIME](#) and [BREACH](#) attacks. CRIME and BREACH attacks can be mitigated in ASP.NET Core with antiforgery tokens. For more information, see [Prevent Cross-Site Request Forgery \(XSRF/CSRF\) attacks in ASP.NET Core](#). For information on mitigating BREACH attacks, see [mitigations](#) at <http://www.breachattack.com/>

Even when `EnableForHttps` is disabled in the app, IIS, IIS Express, and [Azure App Service](#) can apply gzip at the IIS web server. When reviewing response headers, take note of the `Server` value. An unexpected `content-encoding` response header value may be the result of the web server and not the ASP.NET Core app configuration.

When to use Response Compression Middleware

Use server-based response compression technologies in IIS, Apache, or Nginx. The performance of the response compression middleware probably won't match that of the server modules. [HTTP.sys](#) server and [Kestrel](#) server don't currently offer built-in compression support.

Use Response Compression Middleware when the app is:

- Unable to use the following server-based compression technologies:
 - [IIS Dynamic Compression module](#)
 - [Apache mod_deflate module](#)
 - [Nginx Compression and Decompression](#)
- Hosting directly on:
 - [HTTP.sys server](#)
 - [Kestrel server](#)

Response compression

Usually, any response not natively compressed can benefit from response compression. Responses not natively compressed typically include CSS, JavaScript, HTML, XML, and JSON. Don't compress natively compressed assets, such as PNG files. When attempting to further compress a natively compressed response, any small extra reduction in size and transmission time will likely be overshadowed by the time it takes to process the compression. Don't compress files smaller than about 150-1000 bytes, depending on the file's content and the efficiency of compression. The overhead of compressing small files may produce a compressed file larger than the uncompressed file.

When a client can process compressed content, the client must inform the server of its capabilities by sending the [Accept-Encoding](#) header with the request. When a server sends compressed content, it must include information in the [Content-Encoding](#) header on how the compressed response is encoded. Content encoding designations supported by the response compression middleware are shown in the following table.

 Expand table

Accept-Encoding header values	Middleware Supported	Description
br	Yes (default)	Brotli compressed data format
deflate	No	DEFLATE compressed data format
exi	No	W3C Efficient XML Interchange
gzip	Yes	Gzip file format
identity	Yes	"No encoding" identifier: The response must not be encoded.
pack200-gzip	No	Network Transfer Format for Java Archives
*	Yes	Any available content encoding not explicitly requested

For more information, see the [IANA Official Content Coding List](#).

The response compression middleware allows adding additional compression providers for custom `Accept-Encoding` header values. For more information, see [Custom Providers](#) in this article.

The response compression middleware is capable of reacting to quality value (qvalue, `q`) weighting when sent by the client to prioritize compression schemes. For more

information, see [RFC 9110: Accept-Encoding](#).

Compression algorithms are subject to a tradeoff between compression speed and the effectiveness of the compression. *Effectiveness* in this context refers to the size of the output after compression. The smallest size is achieved by the optimal compression.

The headers involved in requesting, sending, caching, and receiving compressed content are described in the following table.

 [Expand table](#)

Header	Role
Accept-Encoding	Sent from the client to the server to indicate the content encoding schemes acceptable to the client.
Content-Encoding	Sent from the server to the client to indicate the encoding of the content in the payload.
Content-Length	When compression occurs, the <code>Content-Length</code> header is removed, since the body content changes when the response is compressed.
Content-MD5	When compression occurs, the <code>Content-MD5</code> header is removed, since the body content has changed and the hash is no longer valid.
Content-Type	Specifies the MIME type of the content. Every response should specify its <code>Content-Type</code> . The response compression middleware checks this value to determine if the response should be compressed. The response compression middleware specifies a set of default MIME types that it can encode, and they can be replaced or added.
Vary	When sent by the server with a value of <code>Accept-Encoding</code> to clients and proxies, the <code>Vary</code> header indicates to the client or proxy that it should cache (vary) responses based on the value of the <code>Accept-Encoding</code> header of the request. The result of returning content with the <code>Vary: Accept-Encoding</code> header is that both compressed and uncompressed responses are cached separately.

Explore the features of the Response Compression Middleware with the [sample app](#).

The sample illustrates:

- The compression of app responses using Gzip and custom compression providers.
- How to add a [MIME type](#) to the default list of MIME types for compression.
- How to add a custom response compression provider.

Configuration

The following code shows how to enable the Response Compression Middleware for default [MIME types](#) and compression providers ([Brotli](#) and [Gzip](#)):

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCompression(options =>
{
    options.EnableForHttps = true;
});

var app = builder.Build();

app.UseResponseCompression();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Notes:

- Setting `EnableForHttps` to `true` is a security risk. See [Compression with HTTPS](#) in this article for more information.
- `app.UseResponseCompression` must be called **before** any middleware that compresses responses. For more information, see [ASP.NET Core Middleware](#).
- Use a tool such as [Firefox Browser Developer](#) to set the `Accept-Encoding` request header and examine the response headers, size, and body.

Submit a request to the sample app without the `Accept-Encoding` header and observe that the response is uncompressed. The `Content-Encoding` header isn't in the Response Headers collection.

For example, in Firefox Developer:

- Select the network tab.
- Right click the request in the [Network request list](#) and select **Edit and resend**
- Change `Accept-Encoding:` from `gzip, deflate, br` to `none`.
- Select **Send**.

Submit a request to the sample app with a browser using the developer tools and observe that the response is compressed. The `Content-Encoding` and `Vary` headers are present on the response.

Providers

Brotli and Gzip compression providers

Use the [BrotliCompressionProvider](#) to compress responses with the [Brotli compressed data format](#).

If [no compression providers are explicitly added](#) to the [CompressionProviderCollection](#):

- The Brotli compression provider and Gzip compression provider are added by default to the array of compression providers.
- Compression defaults to Brotli compression when the Brotli compressed data format is supported by the client. If Brotli isn't supported by the client, compression defaults to Gzip when the client supports Gzip compression.

ⓘ Note

Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see [How to select a version tag of ASP.NET Core source code \(dotnet/AspNetCore.Docs #26205\)](#).

When a compression provider is added, other providers aren't added. For example, if the Gzip compression provider is the only provider explicitly added, no other compression providers are added.

The following code:

- Enables response compression for HTTPS requests.
- Adds the Brotli and Gzip response compression providers.

C#

```
using System.IO.Compression;
using Microsoft.AspNetCore.ResponseCompression;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCompression(options =>
{
    options.EnableForHttps = true;
    options.Providers.Add<BrotliCompressionProvider>();
    options.Providers.Add<GzipCompressionProvider>();
});

builder.Services.Configure<BrotliCompressionProviderOptions>(options =>
```

```

{
    options.Level = CompressionLevel.Fastest;
});

builder.Services.Configure<GzipCompressionProviderOptions>(options =>
{
    options.Level = CompressionLevel.SmallestSize;
});

var app = builder.Build();

app.UseResponseCompression();

app.MapGet("/", () => "Hello World!");

app.Run();

```

Set the compression level with [BrotliCompressionProviderOptions](#) and [GzipCompressionProviderOptions](#). The Brotli and Gzip compression providers default to the fastest compression level, [CompressionLevel.Fastest](#), which might not produce the most efficient compression. If the most efficient compression is desired, configure the response compression middleware for optimal compression.

See [CompressionLevel Enum](#) for values that indicate whether a compression operation emphasizes speed or compression size.

C#

```

using System.IO.Compression;
using Microsoft.AspNetCore.ResponseCompression;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCompression(options =>
{
    options.EnableForHttps = true;
    options.Providers.Add<BrotliCompressionProvider>();
    options.Providers.Add<GzipCompressionProvider>();
});

builder.Services.Configure<BrotliCompressionProviderOptions>(options =>
{
    options.Level = CompressionLevel.Fastest;
});

builder.Services.Configure<GzipCompressionProviderOptions>(options =>
{
    options.Level = CompressionLevel.SmallestSize;
});

var app = builder.Build();

```

```
app.UseResponseCompression();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Custom providers

Create custom compression implementations with [ICompressionProvider](#). The [EncodingName](#) represents the content encoding that this [ICompressionProvider](#) produces. The response compression middleware uses this information to choose the provider based on the list specified in the [Accept-Encoding](#) header of the request.

Requests to the sample app with the [Accept-Encoding: mycustomcompression](#) header return a response with a [Content-Encoding: mycustomcompression](#) header. The client must be able to decompress the custom encoding in order for a custom compression implementation to work.

C#

```
using Microsoft.AspNetCore.ResponseCompression;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCompression(options =>
{
    options.Providers.Add<BrotliCompressionProvider>();
    options.Providers.Add<GzipCompressionProvider>();
    options.Providers.Add<CustomCompressionProvider>();
});

var app = builder.Build();

app.UseResponseCompression();

app.MapGet("/", () => "Hello World!");

app.Run();
```

C#

```
using Microsoft.AspNetCore.ResponseCompression;

public class CustomCompressionProvider : ICompressionProvider
{
    public string EncodingName => "mycustomcompression";
    public bool SupportsFlush => true;
}
```

```

public Stream CreateStream(Stream outputStream)
{
    // Replace with a custom compression stream wrapper.
    return outputStream;
}
}

```

With the preceding code, the response body isn't compressed by the sample. However, the sample shows where to implement a custom compression algorithm.

MIME types

The response compression middleware specifies a default set of MIME types for compression. See the [source code for a complete list of MIME types supported](#).

ⓘ Note

Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see [How to select a version tag of ASP.NET Core source code \(dotnet/AspNetCore.Docs #26205\)](#).

Replace or append MIME types with [ResponseCompressionOptions.MimeTypes](#). Note that wildcard MIME types, such as `text/*` aren't supported. The sample app adds a MIME type for `image/svg+xml` and compresses and serves the ASP.NET Core banner image *banner.svg*.

C#

```

using Microsoft.AspNetCore.ResponseCompression;
using ResponseCompressionSample;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCompression(options =>
{
    options.EnableForHttps = true;
    options.Providers.Add<BrotliCompressionProvider>();
    options.Providers.Add<GzipCompressionProvider>();
    options.Providers.Add<CustomCompressionProvider>();
    options.MimeTypes =
        ResponseCompressionDefaults.MimeTypes.Concat(
            new[] { "image/svg+xml" });
});

```

```
var app = builder.Build();

app.UseResponseCompression();
```

Adding the Vary header

When compressing responses based on the [Accept-Encoding request header](#), there can be uncompressed and multiple compressed versions of the response. In order to instruct client and proxy caches that multiple versions exist and should be stored, the `Vary` header is added with an `Accept-Encoding` value. The response middleware [adds the Vary header](#) automatically when the response is compressed.

ⓘ Note

Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see [How to select a version tag of ASP.NET Core source code \(dotnet/AspNetCore.Docs #26205\)](#).

Middleware issue when behind an Nginx reverse proxy

When a request is proxied by Nginx, the `Accept-Encoding` header is removed. Removal of the `Accept-Encoding` header prevents the response compression middleware from compressing the response. For more information, see [NGINX: Compression and Decompression](#). This issue is tracked by [Figure out pass-through compression for Nginx \(dotnet/aspnetcore#5989\)](#).

Disabling IIS dynamic compression

To disable IIS Dynamic Compression Module configured at the server level, see [Disabling IIS modules](#).

Troubleshoot response compression

Use a tool like [Firefox Browser Developer](#), which allows setting the `Accept-Encoding` request header and study the response headers, size, and body. By default, Response Compression Middleware compresses responses that meet the following conditions:

- The `Accept-Encoding` header is present with a value of `br`, `gzip`, `*`, or custom encoding that matches a custom compression provider. The value must not be `identity` or have a quality value (qvalue, `q`) setting of 0 (zero).
- The MIME type (`Content-Type`) must be set and must match a MIME type configured on the [ResponseCompressionOptions](#).
- The request must not include the `Content-Range` header.
- The request must use insecure protocol (`http`), unless secure protocol (`https`) is configured in the Response Compression Middleware options. *Note the danger described above when enabling secure content compression.*

Azure deployed sample

The sample app deployed to Azure has the following `Program.cs` file:

C#

```
using Microsoft.AspNetCore.ResponseCompression;
using ResponseCompressionSample;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCompression(options =>
{
    options.EnableForHttps = true;
    options.Providers.Add<BrotliCompressionProvider>();
    options.Providers.Add<GzipCompressionProvider>();
    options.Providers.Add<CustomCompressionProvider>();
    options.MimeTypes =
        ResponseCompressionDefaults.MimeTypes.Concat(
            new[] { "image/svg+xml" });
});

var app = builder.Build();

app.UseResponseCompression();

app.Map("/trickle", async (HttpResponse httpResponse) =>
{
    httpResponse.ContentType = "text/plain;charset=utf-8";

    for (int i = 0; i < 20; i++)
    {
        await httpResponse.WriteAsync("a");
        await httpResponse.Body.FlushAsync();
    }
});
```

```

        await Task.Delay(TimeSpan.FromMilliseconds(50));
    }
});

app.Map("/testfile1kb.txt", () => Results.File(

app.Environment.ContentRootFileProvider.GetFileInfo("testfile1kb.txt").PhysicalPath,
    "text/plain;charset=utf-8"));

app.Map("/banner.svg", () => Results.File(

app.Environment.ContentRootFileProvider.GetFileInfo("banner.svg").PhysicalPath,
    "image/svg+xml;charset=utf-8"));

app.MapFallback(() => LoremIpsum.Text);

app.Run();

```

Additional resources

ⓘ Note

Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see [How to select a version tag of ASP.NET Core source code \(dotnet/AspNetCore.Docs #26205\)](#).[↗]

- [View or download sample code](#)[↗] (how to download)
- [Response compression middleware source](#)[↗]
- [ASP.NET Core Middleware](#)
- [Mozilla Developer Network: Accept-Encoding](#)[↗]
- [RFC 9110 Section 8.4.1: Content Codings](#)[↗]
- [RFC 9110 Section 8.4.1.3: Gzip Coding](#)[↗]
- [GZIP file format specification version 4.3](#)[↗]