ASP.NET Core Performance Best Practices

Article • 06/04/2022 • 18 minutes to read

By Mike Rousos

This article provides guidelines for performance best practices with ASP.NET Core.

Cache aggressively

Caching is discussed in several parts of this document. For more information, see Response caching in ASP.NET Core.

Understand hot code paths

In this document, a *hot code path* is defined as a code path that is frequently called and where much of the execution time occurs. Hot code paths typically limit app scale-out and performance and are discussed in several parts of this document.

Avoid blocking calls

ASP.NET Core apps should be designed to process many requests simultaneously. Asynchronous APIs allow a small pool of threads to handle thousands of concurrent requests by not waiting on blocking calls. Rather than waiting on a long-running synchronous task to complete, the thread can work on another request.

A common performance problem in ASP.NET Core apps is blocking calls that could be asynchronous. Many synchronous blocking calls lead to Thread Pool starvation and degraded response times.

Do not:

- Block asynchronous execution by calling Task.Wait or Task<TResult>.Result.
- Acquire locks in common code paths. ASP.NET Core apps are most performant when architected to run code in parallel.
- Call Task.Run and immediately await it. ASP.NET Core already runs app code on normal Thread Pool threads, so calling Task.Run only results in extra unnecessary Thread Pool scheduling. Even if the scheduled code would block a thread, Task.Run does not prevent that.

Do:

- Make hot code paths asynchronous.
- Call data access, I/O, and long-running operations APIs asynchronously if an asynchronous API is available. Do **not** use Task.Run to make a synchronous API asynchronous.
- Make controller/Razor Page actions asynchronous. The entire call stack is asynchronous in order to benefit from async/await patterns.

A profiler, such as PerfView , can be used to find threads frequently added to the Thread Pool. The `Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Start` event indicates a thread added to the thread pool.

Return large collections across multiple smaller pages

A webpage shouldn't load large amounts of data all at once. When returning a collection of objects, consider whether it could lead to performance issues. Determine if the design could produce the following poor outcomes:

- OutOfMemoryException or high memory consumption
- Thread pool starvation (see the following remarks on IAsyncEnumerable<T>)
- Slow response times
- Frequent garbage collection

Do add pagination to mitigate the preceding scenarios. Using page size and page index parameters, developers should favor the design of returning a partial result. When an exhaustive result is required, pagination should be used to asynchronously populate batches of results to avoid locking server resources.

For more information on paging and limiting the number of returned records, see:

- Performance considerations
- Add paging to an ASP.NET Core app

Return `IEnumerable<T>` or `IAsyncEnumerable<T>`

Returning `IEnumerable<T>` from an action results in synchronous collection iteration by the serializer. The result is the blocking of calls and a potential for thread pool starvation. To avoid synchronous enumeration, use `ToListAsync` before returning the enumerable.

Beginning with ASP.NET Core 3.0, `IAsyncEnumerable<T>` can be used as an alternative to `IEnumerable<T>` that enumerates asynchronously. For more information, see Controller action return types.

Minimize large object allocations

The .NET Core garbage collector manages allocation and release of memory automatically in ASP.NET Core apps. Automatic garbage collection generally means that developers don't need to worry about how or when memory is freed. However, cleaning up unreferenced objects takes CPU time, so developers should minimize allocating objects in hot code paths. Garbage collection is especially expensive on large objects (> 85 K bytes). Large objects are stored on the large object heap and require a full (generation 2) garbage collection to clean up. Unlike generation 0 and generation 1 collections, a generation 2 collection requires a temporary suspension of app execution. Frequent allocation and de-allocation of large objects can cause inconsistent performance.

Recommendations:

- **Do** consider caching large objects that are frequently used. Caching large objects prevents expensive allocations.
- **Do** pool buffers by using an ArrayPool<T> to store large arrays.
- **Do not** allocate many, short-lived large objects on hot code paths.

Memory issues, such as the preceding, can be diagnosed by reviewing garbage collection (GC) stats in PerfView and examining:

- Garbage collection pause time.
- What percentage of the processor time is spent in garbage collection.
- How many garbage collections are generation 0, 1, and 2.

For more information, see Garbage Collection and Performance.

Optimize data access and I/O

Interactions with a data store and other remote services are often the slowest parts of an ASP.NET Core app. Reading and writing data efficiently is critical for good performance.

Recommendations:

- **Do** call all data access APIs asynchronously.

- **Do not** retrieve more data than is necessary. Write queries to return just the data that's necessary for the current HTTP request.
- **Do** consider caching frequently accessed data retrieved from a database or remote service if slightly out-of-date data is acceptable. Depending on the scenario, use a MemoryCache or a DistributedCache. For more information, see Response caching in ASP.NET Core.
- **Do** minimize network round trips. The goal is to retrieve the required data in a single call rather than several calls.
- **Do** use no-tracking queries in Entity Framework Core when accessing data for read-only purposes. EF Core can return the results of no-tracking queries more efficiently.
- **Do** filter and aggregate LINQ queries (with `.Where`, `.Select`, or `.Sum` statements, for example) so that the filtering is performed by the database.
- **Do** consider that EF Core resolves some query operators on the client, which may lead to inefficient query execution. For more information, see Client evaluation performance issues.
- **Do not** use projection queries on collections, which can result in executing "N + 1" SQL queries. For more information, see Optimization of correlated subqueries.

See EF High Performance for approaches that may improve performance in high-scale apps:

- DbContext pooling
- Explicitly compiled queries

We recommend measuring the impact of the preceding high-performance approaches before committing the code base. The additional complexity of compiled queries may not justify the performance improvement.

Query issues can be detected by reviewing the time spent accessing data with Application Insights or with profiling tools. Most databases also make statistics available concerning frequently executed queries.

Pool HTTP connections with HttpClientFactory

Although HttpClient implements the `IDisposable` interface, it's designed for reuse. Closed `HttpClient` instances leave sockets open in the `TIME_WAIT` state for a short period of time. If a code path that creates and disposes of `HttpClient` objects is frequently used, the app may exhaust available sockets. HttpClientFactory was introduced in ASP.NET Core 2.1 as a solution to this problem. It handles pooling HTTP connections to optimize performance and reliability.

Recommendations:

- **Do not** create and dispose of `HttpClient` instances directly.
- **Do** use HttpClientFactory to retrieve `HttpClient` instances. For more information, see Use HttpClientFactory to implement resilient HTTP requests.

Keep common code paths fast

You want all of your code to be fast. Frequently-called code paths are the most critical to optimize. These include:

- Middleware components in the app's request processing pipeline, especially middleware run early in the pipeline. These components have a large impact on performance.
- Code that's executed for every request or multiple times per request. For example, custom logging, authorization handlers, or initialization of transient services.

Recommendations:

- **Do not** use custom middleware components with long-running tasks.
- **Do** use performance profiling tools, such as Visual Studio Diagnostic Tools or PerfView), to identify hot code paths.

Complete long-running Tasks outside of HTTP requests

Most requests to an ASP.NET Core app can be handled by a controller or page model calling necessary services and returning an HTTP response. For some requests that involve long-running tasks, it's better to make the entire request-response process asynchronous.

Recommendations:

- **Do not** wait for long-running tasks to complete as part of ordinary HTTP request processing.
- **Do** consider handling long-running requests with background services or out of process with an Azure Function. Completing work out-of-process is especially beneficial for CPU-intensive tasks.
- **Do** use real-time communication options, such as SignalR, to communicate with clients asynchronously.

Minify client assets

ASP.NET Core apps with complex front-ends frequently serve many JavaScript, CSS, or image files. Performance of initial load requests can be improved by:

- Bundling, which combines multiple files into one.
- Minifying, which reduces the size of files by removing whitespace and comments.

Recommendations:

- **Do** use the bundling and minification guidelines, which mentions compatible tools and shows how to use ASP.NET Core's `environment` tag to handle both `Development` and `Production` environments.
- **Do** consider other third-party tools, such as Webpack , for complex client asset management.

Compress responses

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. For more information, see Response compression.

Use the latest ASP.NET Core release

Each new release of ASP.NET Core includes performance improvements. Optimizations in .NET Core and ASP.NET Core mean that newer versions generally outperform older versions. For example, .NET Core 2.1 added support for compiled regular expressions and benefitted from Span<T>. ASP.NET Core 2.2 added support for HTTP/2. ASP.NET Core 3.0 adds many improvements that reduce memory usage and improve throughput. If performance is a priority, consider upgrading to the current version of ASP.NET Core.

Minimize exceptions

Exceptions should be rare. Throwing and catching exceptions is slow relative to other code flow patterns. Because of this, exceptions shouldn't be used to control normal program flow.

Recommendations:

- **Do not** use throwing or catching exceptions as a means of normal program flow, especially in hot code paths.
- **Do** include logic in the app to detect and handle conditions that would cause an exception.
- **Do** throw or catch exceptions for unusual or unexpected conditions.

App diagnostic tools, such as Application Insights, can help to identify common exceptions in an app that may affect performance.

Performance and reliability

The following sections provide performance tips and known reliability problems and solutions.

Avoid synchronous read or write on HttpRequest/HttpResponse body

All I/O in ASP.NET Core is asynchronous. Servers implement the `Stream` interface, which has both synchronous and asynchronous overloads. The asynchronous ones should be preferred to avoid blocking thread pool threads. Blocking threads can lead to thread pool starvation.

Do not do this: The following example uses the ReadToEnd. It blocks the current thread to wait for the result. This is an example of sync over async .

C#

```
public class BadStreamReaderController : Controller
{
    [HttpGet("/contoso")]
    public ActionResult<ContosoData> Get()
    {
        var json = new StreamReader(Request.Body).ReadToEnd();

        return JsonSerializer.Deserialize<ContosoData>(json);
    }
}
```

In the preceding code, `Get` synchronously reads the entire HTTP request body into memory. If the client is slowly uploading, the app is doing sync over async. The app does sync over async because Kestrel does **NOT** support synchronous reads.

Do this: The following example uses ReadToEndAsync and does not block the thread while reading.

C#

```
public class GoodStreamReaderController : Controller
{
    [HttpGet("/contoso")]
    public async Task<ActionResult<ContosoData>> Get()
    {
        var json = await new StreamReader(Request.Body).ReadToEndAsync();

        return JsonSerializer.Deserialize<ContosoData>(json);
    }

}
```

The preceding code asynchronously reads the entire HTTP request body into memory.

> ⚠ **Warning**
>
> If the request is large, reading the entire HTTP request body into memory could lead to an out of memory (OOM) condition. OOM can result in a Denial Of Service. For more information, see **Avoid reading large request bodies or response bodies into memory** in this document.

Do this: The following example is fully asynchronous using a non buffered request body:

C#

```
public class GoodStreamReaderController : Controller
{
    [HttpGet("/contoso")]
    public async Task<ActionResult<ContosoData>> Get()
    {
        return await JsonSerializer.DeserializeAsync<ContosoData>
(Request.Body);
    }
}
```

The preceding code asynchronously de-serializes the request body into a C# object.

Prefer ReadFormAsync over Request.Form

Use `HttpContext.Request.ReadFormAsync` instead of `HttpContext.Request.Form`. `HttpContext.Request.Form` can be safely read only with the following conditions:

- The form has been read by a call to `ReadFormAsync`, and
- The cached form value is being read using `HttpContext.Request.Form`

Do not do this: The following example uses `HttpContext.Request.Form`. `HttpContext.Request.Form` uses sync over async and can lead to thread pool starvation.

C#

```
public class BadReadController : Controller
{
    [HttpPost("/form-body")]
    public IActionResult Post()
    {
        var form =  HttpContext.Request.Form;

        Process(form["id"], form["name"]);

        return Accepted();
    }
```

Do this: The following example uses `HttpContext.Request.ReadFormAsync` to read the form body asynchronously.

C#

```
public class GoodReadController : Controller
{
    [HttpPost("/form-body")]
    public async Task<IActionResult> Post()
    {
       var form = await HttpContext.Request.ReadFormAsync();

        Process(form["id"], form["name"]);

        return Accepted();
    }
```

Avoid reading large request bodies or response bodies into memory

In .NET, every object allocation greater than 85 KB ends up in the large object heap (LOH). Large objects are expensive in two ways:

- The allocation cost is high because the memory for a newly allocated large object has to be cleared. The CLR guarantees that memory for all newly allocated objects is cleared.
- LOH is collected with the rest of the heap. LOH requires a full garbage collection or Gen2 collection.

This blog post describes the problem succinctly:

> When a large object is allocated, it's marked as Gen 2 object. Not Gen 0 as for small objects. The consequences are that if you run out of memory in LOH, GC cleans up the whole managed heap, not only LOH. So it cleans up Gen 0, Gen 1 and Gen 2 including LOH. This is called full garbage collection and is the most time-consuming garbage collection. For many applications, it can be acceptable. But definitely not for high-performance web servers, where few big memory buffers are needed to handle an average web request (read from a socket, decompress, decode JSON & more).

Naively storing a large request or response body into a single `byte[]` or `string`:

- May result in quickly running out of space in the LOH.
- May cause performance issues for the app because of full GCs running.

Working with a synchronous data processing API

When using a serializer/de-serializer that only supports synchronous reads and writes (for example, Json.NET):

- Buffer the data into memory asynchronously before passing it into the serializer/de-serializer.

> **⚠ Warning**
>
> If the request is large, it could lead to an out of memory (OOM) condition. OOM can result in a Denial Of Service. For more information, see **Avoid reading large request bodies or response bodies into memory** in this document.

ASP.NET Core 3.0 uses System.Text.Json by default for JSON serialization. System.Text.Json:

- Reads and writes JSON asynchronously.
- Is optimized for UTF-8 text.
- Typically higher performance than `Newtonsoft.Json`.

Do not store IHttpContextAccessor.HttpContext in a field

The IHttpContextAccessor.HttpContext returns the `HttpContext` of the active request when accessed from the request thread. The `IHttpContextAccessor.HttpContext` should **not** be stored in a field or variable.

Do not do this: The following example stores the `HttpContext` in a field and then attempts to use it later.

C#

```
public class MyBadType
{
    private readonly HttpContext _context;
    public MyBadType(IHttpContextAccessor accessor)
    {
        _context = accessor.HttpContext;
    }

    public void CheckAdmin()
    {
        if (!_context.User.IsInRole("admin"))
        {
            throw new UnauthorizedAccessException("The current user isn't an
admin");
        }
    }
}
```

The preceding code frequently captures a null or incorrect `HttpContext` in the constructor.

Do this: The following example:

- Stores the IHttpContextAccessor in a field.
- Uses the `HttpContext` field at the correct time and checks for `null`.

C#

```
public class MyGoodType
{
    private readonly IHttpContextAccessor _accessor;
    public MyGoodType(IHttpContextAccessor accessor)
    {
        _accessor = accessor;
    }

    public void CheckAdmin()
    {
        var context = _accessor.HttpContext;
        if (context != null && !context.User.IsInRole("admin"))
        {
            throw new UnauthorizedAccessException("The current user isn't an
admin");
        }
    }
}
```

Do not access HttpContext from multiple threads

`HttpContext` is *NOT* thread-safe. Accessing `HttpContext` from multiple threads in parallel can result in undefined behavior such as hangs, crashes, and data corruption.

Do not do this: The following example makes three parallel requests and logs the incoming request path before and after the outgoing HTTP request. The request path is accessed from multiple threads, potentially in parallel.

C#

```
public class AsyncBadSearchController : Controller
{
    [HttpGet("/search")]
    public async Task<SearchResults> Get(string query)
    {
        var query1 = SearchAsync(SearchEngine.Google, query);
        var query2 = SearchAsync(SearchEngine.Bing, query);
        var query3 = SearchAsync(SearchEngine.DuckDuckGo, query);

        await Task.WhenAll(query1, query2, query3);

        var results1 = await query1;
        var results2 = await query2;
        var results3 = await query3;

        return SearchResults.Combine(results1, results2, results3);
```

```
    }

    private async Task<SearchResults> SearchAsync(SearchEngine engine,
string query)
    {
        var searchResults = _searchService.Empty();
        try
        {
            _logger.LogInformation("Starting search query from {path}.",
                                   HttpContext.Request.Path);
            searchResults = _searchService.Search(engine, query);
            _logger.LogInformation("Finishing search query from {path}.",
                                   HttpContext.Request.Path);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed query from {path}",
                             HttpContext.Request.Path);
        }

        return await searchResults;
    }
```

Do this: The following example copies all data from the incoming request before making the three parallel requests.

C#

```
public class AsyncGoodSearchController : Controller
{
    [HttpGet("/search")]
    public async Task<SearchResults> Get(string query)
    {
        string path = HttpContext.Request.Path;
        var query1 = SearchAsync(SearchEngine.Google, query,
                                 path);
        var query2 = SearchAsync(SearchEngine.Bing, query, path);
        var query3 = SearchAsync(SearchEngine.DuckDuckGo, query, path);

        await Task.WhenAll(query1, query2, query3);

        var results1 = await query1;
        var results2 = await query2;
        var results3 = await query3;

        return SearchResults.Combine(results1, results2, results3);
    }

    private async Task<SearchResults> SearchAsync(SearchEngine engine,
string query,
                                                  string path)
    {
        var searchResults = _searchService.Empty();
```

```
        try
        {
            _logger.LogInformation("Starting search query from {path}.",
                                   path);
            searchResults = await _searchService.SearchAsync(engine, query);
            _logger.LogInformation("Finishing search query from {path}.",
path);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed query from {path}", path);
        }

        return await searchResults;
    }
```

Do not use the HttpContext after the request is complete

`HttpContext` is only valid as long as there is an active HTTP request in the ASP.NET Core pipeline. The entire ASP.NET Core pipeline is an asynchronous chain of delegates that executes every request. When the `Task` returned from this chain completes, the `HttpContext` is recycled.

Do not do this: The following example uses `async void` which makes the HTTP request complete when the first `await` is reached:

- Which is **ALWAYS** a bad practice in ASP.NET Core apps.
- Accesses the `HttpResponse` after the HTTP request is complete.
- Crashes the process.

C#

```
public class AsyncBadVoidController : Controller
{
    [HttpGet("/async")]
    public async void Get()
    {
        await Task.Delay(1000);

        // The following line will crash the process because of writing af-
ter the
        // response has completed on a background thread. Notice async void
Get()

        await Response.WriteAsync("Hello World");
```

```
    }
}
```

Do this: The following example returns a `Task` to the framework, so the HTTP request doesn't complete until the action completes.

C#

```
public class AsyncGoodTaskController : Controller
{
    [HttpGet("/async")]
    public async Task Get()
    {
        await Task.Delay(1000);

        await Response.WriteAsync("Hello World");
    }
}
```

Do not capture the HttpContext in background threads

Do not do this: The following example shows a closure is capturing the `HttpContext` from the `Controller` property. This is a bad practice because the work item could:

- Run outside of the request scope.
- Attempt to read the wrong `HttpContext`.

C#

```
[HttpGet("/fire-and-forget-1")]
public IActionResult BadFireAndForget()
{
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);

        var path = HttpContext.Request.Path;
        Log(path);
    });

    return Accepted();
}
```

Do this: The following example:

- Copies the data required in the background task during the request.
- Doesn't reference anything from the controller.

C#

```
[HttpGet("/fire-and-forget-3")]
public IActionResult GoodFireAndForget()
{
    string path = HttpContext.Request.Path;
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);

        Log(path);
    });

    return Accepted();
}
```

Background tasks should be implemented as hosted services. For more information, see Background tasks with hosted services.

Do not capture services injected into the controllers on background threads

Do not do this: The following example shows a closure is capturing the `DbContext` from the `Controller` action parameter. This is a bad practice. The work item could run outside of the request scope. The `ContosoDbContext` is scoped to the request, resulting in an `ObjectDisposedException`.

C#

```
[HttpGet("/fire-and-forget-1")]
public IActionResult FireAndForget1([FromServices]ContosoDbContext context)
{
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);

        context.Contoso.Add(new Contoso());
        await context.SaveChangesAsync();
    });

    return Accepted();
}
```

Do this: The following example:

- Injects an IServiceScopeFactory in order to create a scope in the background work item. `IServiceScopeFactory` is a singleton.
- Creates a new dependency injection scope in the background thread.
- Doesn't reference anything from the controller.
- Doesn't capture the `ContosoDbContext` from the incoming request.

C#

```
[HttpGet("/fire-and-forget-3")]
public IActionResult FireAndForget3([FromServices]IServiceScopeFactory
                                    serviceScopeFactory)
{
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);

        using (var scope = serviceScopeFactory.CreateScope())
        {
            var context =
scope.ServiceProvider.GetRequiredService<ContosoDbContext>();

            context.Contoso.Add(new Contoso());

            await context.SaveChangesAsync();
        }
    });

    return Accepted();
}
```

The following highlighted code:

- Creates a scope for the lifetime of the background operation and resolves services from it.
- Uses `ContosoDbContext` from the correct scope.

C#

```
[HttpGet("/fire-and-forget-3")]
public IActionResult FireAndForget3([FromServices]IServiceScopeFactory
                                    serviceScopeFactory)
{
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);
```

```
        using (var scope = serviceScopeFactory.CreateScope())
        {
            var context =
scope.ServiceProvider.GetRequiredService<ContosoDbContext>();

            context.Contoso.Add(new Contoso());

            await context.SaveChangesAsync();
        }
    });

    return Accepted();
}
```

Do not modify the status code or headers after the response body has started

ASP.NET Core does not buffer the HTTP response body. The first time the response is written:

- The headers are sent along with that chunk of the body to the client.
- It's no longer possible to change response headers.

Do not do this: The following code tries to add response headers after the response has already started:

C#

```
app.Use(async (context, next) =>
{
    await next();

    context.Response.Headers["test"] = "test value";
});
```

In the preceding code, `context.Response.Headers["test"] = "test value";` will throw an exception if `next()` has written to the response.

Do this: The following example checks if the HTTP response has started before modifying the headers.

C#

```
app.Use(async (context, next) =>
{
```

```
    await next();

    if (!context.Response.HasStarted)
    {
        context.Response.Headers["test"] = "test value";
    }
});
```

Do this: The following example uses `HttpResponse.OnStarting` to set the headers before the response headers are flushed to the client.

Checking if the response has not started allows registering a callback that will be invoked just before response headers are written. Checking if the response has not started:

- Provides the ability to append or override headers just in time.
- Doesn't require knowledge of the next middleware in the pipeline.

C#

```
app.Use(async (context, next) =>
{
    context.Response.OnStarting(() =>
    {
        context.Response.Headers["someheader"] = "somevalue";
        return Task.CompletedTask;
    });

    await next();
});
```

Do not call next() if you have already started writing to the response body

Components only expect to be called if it's possible for them to handle and manipulate the response.

Use In-process hosting with IIS

Using in-process hosting, an ASP.NET Core app runs in the same process as its IIS worker process. In-process hosting provides improved performance over out-of-process hosting because requests aren't proxied over the loopback adapter. The loopback adapter is a network interface that returns outgoing network traffic back to the same

machine. IIS handles process management with the Windows Process Activation Service (WAS).

Projects default to the in-process hosting model in ASP.NET Core 3.0 and later.

For more information, see Host ASP.NET Core on Windows with IIS