8

Enhancing Applications with Middleware in ASP.NET Core 9

ASP.NET Core 9 offers a robust and flexible framework designed to handle high-demand web applications. A key component of this framework is middleware, which allows developers to interact directly with the request and response pipeline. Understanding and leveraging middleware can significantly enhance your application's capabilities. This chapter will dive deep into middleware, exploring its structure, implementation, and practical applications, such as global error handling, request limiting, and more.

In this chapter, we will focus on the following topics:

- Knowing the middleware pipeline
- Implementing custom middleware
- Working with factory-based middleware
- Adding capabilities to applications using middleware
- Creating an extension method for middleware registration

In this chapter, we will explore essential best practices for developing applications with ASP.NET Core 9, covering the correct use of asynchronous mechanisms, HTTP requests, and application instrumentation through logs.

Technical requirements

To support the learning of this chapter, the following tools must be present in your development environment:

- **Docker**: The Docker engine must be installed on your operating system and have an SQL Server container running. You can find more details about Docker and SQL Server containers in *Chapter 5*.

- **Postman**: This tool will be used to execute requests to APIs of the developed application.
- **Redis Insight**: This tool is used to connect to a Redis Server database (`https://redis.io/insight/`).

The code examples used in this chapter can be found in the book's GitHub repository: `https://github.com/PacktPublishing/ASP.NET-Core-9.0-Essentials/tree/main/Chapter08`

Knowing the middleware pipeline

During the previous chapters, we used several features of ASP.NET Core 9, including middleware.

Middleware is a pipeline model used during the execution flow of an ASP.NET Core 9 web application to handle requests and responses, and the applications developed in this book already use some standard middleware from the .NET platform, such as **Authentication**, **Authorization**, **Cross-Origin Resource Sharing** (**CORS**), and so on.

The ASP.NET Core request pipeline consists of a sequence of request delegates, called one after the other. *Figure 8.1* demonstrates the concept:

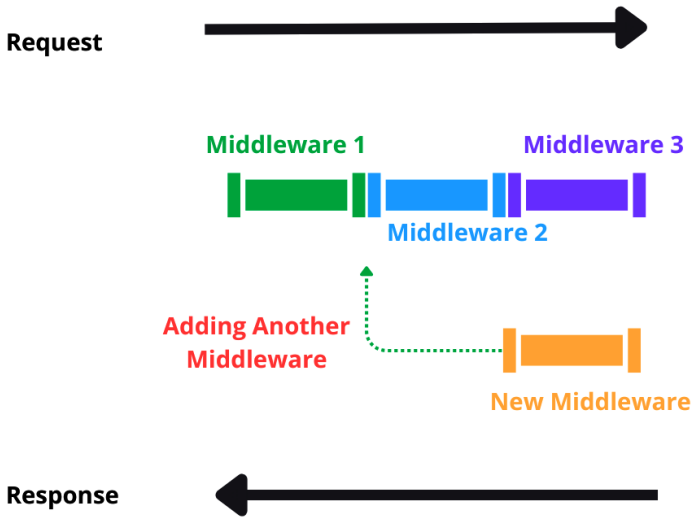

Figure 8.1 – ASP.NET Core 9 middleware pipeline

Request delegates are configured using the `Run`, `Map`, and `Use` extension methods typically configured in the `Program.cs` file.

Each extension method has a template for registering a request delegate:

- `Run`: The `app.Run` method is used to define an inline middleware that handles the request and completes the response, as in the following example code that implements an inline middleware:

```
app.Run(async context =>
{
  await context.Response.WriteAsync("Hello Inline middleware!");
});
```

- `Map`: The `app.Map` method is used to create a branch in the middleware pipeline. In the following code, requests to `/SomeRoute` are handled by this middleware branch. The middleware in the branch writes a message to the response:

```
app.Map("/SomeRoute", someRouteApp =>
{
  someRouteApp.Use(async (context, next) =>
  {
    Console.WriteLine("In SomeRoute middleware");
    await context.Response.WriteAsync("Hello from the SomeRoute
middleware!");
  });
});
```

- `Use`: The `app.Use` method is used to add middleware to the pipeline. The following code uses a middleware to log the request method and path before calling the next middleware in the pipeline. After the next middleware completes, it logs the response status code:

```
app.Use(async (context, next) =>
{
  // Log the request
  Console.WriteLine($"Request:
    {context.Request.Method}
    {context.Request.Path}");
  await next.Invoke();
  // Log the response
  Console.WriteLine($"Response:
    {context.Response.StatusCode}");
});
```

The use of middleware brings constant benefits to applications; we will understand in greater detail the use of different approaches, such as the creation of middleware classes.

Now, let's learn about how the middleware execution flow works.

Understanding middleware flow

When the application receives a request, it goes through each middleware component in the order they are registered, and the following cases can be executed:

- **Process the request and pass it to the next piece of middleware**: It's like a relay race where each runner passes the baton to the next. Each piece of middleware does its part and then calls the next one in line to continue processing the request.
- **Process the request and break the chain, preventing other middleware from running**: Imagine a security checkpoint at an airport. If security finds a problem, they may stop you for additional checks, preventing you from proceeding. Likewise, the middleware may decide to handle the request completely and stop further processing.
- **Process the response as it moves up the chain**: This is like sending a package through multiple stages of inspection. Once the package reaches the final stage, it is inspected again at each stage on the way back, ensuring that everything is in order before being delivered.

The layered approach allows for powerful and flexible handling of HTTP requests and responses. Middleware can be used for a variety of tasks, such as logging, authentication, error handling, and more.

Furthermore, the order in which you register middleware is crucial, as it defines the flow of the request and response pipeline. We can see a representation of the middleware execution flow in *Figure 8.2*:

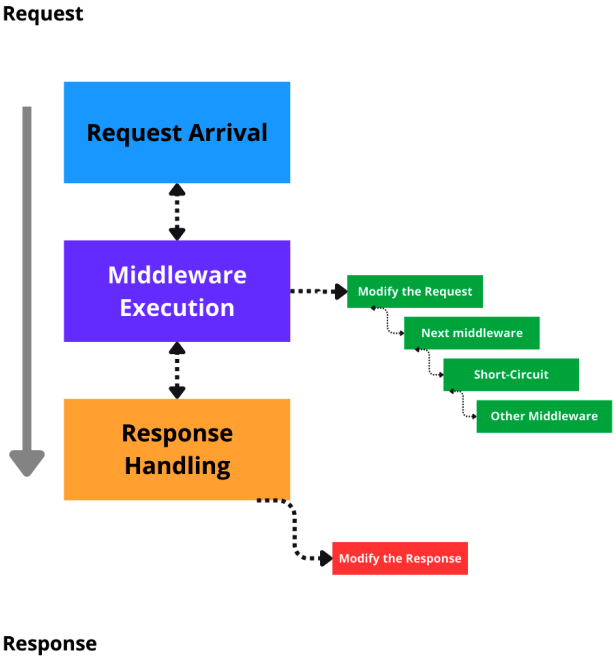

Figure 8.2 – Middleware execution flow

Let's see how the flow works in detail:

- **Request arrival**: When a request arrives at the server, it enters the pipeline and reaches the first middleware component
- **Middleware execution**: Each middleware can do the following:
  - **Modify the request**: Middleware can change aspects of the request, such as adding headers or changing the request path
  - **Move to next middleware**: After processing, the middleware can call the next middleware in the pipeline using `await next`, which we'll discuss in the *Implementing custom middleware* section
- **Short-circuiting the pipeline**: The middleware may decide not to call the next middleware, effectively ending request processing early
- **Response handling**: When the request reaches the end of the pipeline, the response goes back through the middleware components in reverse order
- **Modifying the response**: The middleware can change the response, such as adding headers, changing the status code, or logging information

Middleware order

ASP.NET Core 9 has, by default, some middleware available to handle requests and responses. However, the order in which this middleware is inserted completely changes the application's execution flow and, in some cases, can cause malfunctions. For example, it is important to add middleware authentication before middleware authorization; otherwise, how can you validate authorization without being authenticated?

In any case, in addition to standard middleware, there is an order of execution for customized middleware.

To learn more about middleware order, see the following link: `https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-9.0#middleware-order`.

By working on the middleware execution flow, we have the ability to add several powerful possibilities to our applications, and we will learn some more benefits in the next section.

Benefits of middleware and best practices

Middleware plays a key role in ASP.NET Core 9 applications, offering a number of benefits that contribute to the robustness, maintainability, and extensibility of your application. Understanding these benefits allows you to use this resource effectively, so let's look at this in more detail:

- **Modularity**: Modularity means that middleware is an independent unit of functionality that can be easily added, removed, or replaced without affecting the rest of the application. This modularity allows developers to create reusable middleware components that can be shared between different projects or in different parts of the same project.
- **Composition**: Middleware can be composed in multiple orders to achieve different behaviors. This compositional nature allows you to tailor the request and response pipeline to the specific needs of your application.

  Let's say you have three middleware components: one for logging, one for authentication, and one for handling errors. You can compose these middleware components in the desired order:

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

app.UseMiddleware<ErrorHandlingMiddleware>();
app.UseMiddleware<AuthenticationMiddleware>();
app.UseMiddleware<RequestLoggingMiddleware>();

app.Run(async context =>
{
    await context.Response.WriteAsync("Hello,
      World!");
});

app.Run();
```

  As you can see in the preceding code, the `app.UseMiddleware` method adds middleware to handle errors, authentication, and logging in the application. The `app.Run` method just creates a standard request response, returning a `Hello World` message.

  It is important to consider the following factors:

  - If you want to rearrange the order of these middleware components, the way requests are processed and errors are handled will be different.
  - **SoC (Separation of concerns)**: Middleware allows for a clear separation of different concerns, enabling the definition of a clear execution context in a pipeline and facilitating a clearer, extensible, and maintainable code base.

- **Extensibility**: You can develop a custom middleware to extend the application's functionality – for example, by adding validation capabilities to requests or modifying responses globally within the application.

Suppose you need custom middleware to validate an API key in request headers. You can create this middleware as follows:

```
public class ApiKeyCheckMiddleware
{
    private readonly RequestDelegate _next;
    private const string API_KEY = "X-API-KEY";
    private const string VALID_API_KEY = "XYZ123";

    public ApiKeyCheckMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        if (!context.Request.Headers
          .TryGetValue(API_KEY,
          out var extractedApiKey) ||
          extractedApiKey != VALID_API_KEY)
        {
            context.Response.StatusCode = 401;
            await context.Response
              .WriteAsync("Unauthorized");
            return;
        }
        await _next(context);
    }
}
```

The preceding code aims to create a customized middleware that checks the existence of an API key that must be provided in the header of a request, where the header key is `X-API-KEY` and the expected value is, exactly, `XYZ123`.

When performing validation, if the header and value are not part of the request, then the user receives an `unauthorized` return message with HTTP status code `401`.

In fact, middleware is a powerful feature that allows you to have greater control over the flow of requests and responses in an application developed in ASP.NET Core 9.

Don't worry about the details related to the preceding code examples. We will learn about the structure of a middleware class in the *Implementing custom middleware* section.

Despite the great benefits of applications using middleware, it is important to be aware of good practices; otherwise, what could be a benefit could become a major problem.

Let's look at some best practices:

- **Order matters**: The order in which middleware components are added is crucial as it affects how requests and responses are processed.
- **Keep it simple**: Middleware should do one thing and do it well. Complex logic should be avoided in middleware.
- **Error handling**: Make sure your middleware components handle exceptions and errors in the same way as other classes in your application.
- **Performance**: Be aware of the impact of middleware on performance, especially in high-load scenarios. As it operates in request and response processes, avoid large amounts of processing during these stages to avoid causing problems for users and the application.
- **Reuse existing middleware**: Use built-in middleware whenever possible to reduce the need for custom implementations. As we have already learned, there is some middleware available in ASP.NET Core 9.

Now that we understand the principles, benefits, and best practices of middleware, let's implement our first custom middleware and learn the details of this approach.

Implementing custom middleware

Custom middleware allows you to encapsulate functionality and reuse it in different parts of your application.

Creating custom middleware in ASP.NET Core 9 involves several steps, such as the following:

- Middleware class definition
- The implementation of the `Invoke` or `InvokeAsync` method
- Middleware registration in the request pipeline

Let's analyze the following code, which represents a customized middleware:

```
public class BeforeAfterRequestMiddleware
{
  private readonly RequestDelegate _next;

  public BeforeAfterRequestMiddleware(RequestDelegate next)
  {
```

```
    _next = next;
    }

    public async Task InvokeAsync(HttpContext context)
    {
      // Logging request information
      Console.WriteLine($"Request:
        {context.Request.Method}
        {context.Request.Path}");

      // Call the next middleware in the pipeline
      await _next(context);

      // Logging response information
      Console.WriteLine($"Response:
        {context.Response.StatusCode}");
    }
}
```

This custom middleware code aims to just write a string to the console at the beginning and in the response of the request.

However, it is important to understand the structure of the preceding code:

- `RequestDelegate`: This is a delegate that represents the next middleware in the pipeline. This delegate is stored in a field called `_next` for use in the context of the class.
- `Constructor`: The class constructor receives an instance of the `RequestDelegate` class as a parameter, representing the next middleware in the execution flow.
- `Invoke` or `InvokeAsync` method: Contains the logic for processing HTTP requests. The difference between the methods is that one is executed asynchronously and the other is not. The `InvokeAsync` method receives an `HttpContext` object as a parameter. The `HttpContext` object allows you to access request and response information. It is a good practice to use the `InvokeAsync` method to improve performance and scalability.
- `await _next(context)`: Execution of the `_next` delegate, which receives the `HttpContext` object as a parameter. In this example, we are just writing a string containing request information before propagating the execution of the next middleware, and then another string is written with response information after executing the middleware.

Dependency injection (DI) in middleware

Custom middleware classes must use the **Explicit Dependencies Principle** (**EDP**), as we have already learned in previous chapters, where the dependencies of a class are defined in the constructor.

As middleware is built during application initialization, it is not possible to inject services added to a scoped lifetime as is done with each request in a `Controller` class, for example.

So, if you want to use any services available in DI control in a middleware class, add these services to the signature of the `InvokeAsync` method, which can accept additional parameters resolved by DI.

The previous code example, although simple, demonstrates the basic structure of a middleware, which requires a `RequestDelegated` field, a constructor that depends on an instance of `RequestDelegated`, and the implementation of the `Invoke` or `InvokeAsync` method.

For the customized middleware to be used in the application, it is necessary to register it in the ASP.NET Core 9 execution pipeline through the `Program.cs` file:

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

app.UseMiddleware<BeforeAfterRequestMiddleware>();
app.Run(async context =>
{
    await context.Response.WriteAsync("Hello, World!");
});

app.Run();
```

The preceding code has been shortened to make it easier to read and learn. For the middleware to be registered, the `UseMiddleware` extension method is used, which is a generic method, where we define the previously executed custom middleware as the type.

During the application startup flow, all custom or non-customized middleware is created, forming part of the application lifecycle, and not by request, as is generally done in scoped services. This behavior prevents other dependencies from being added to the constructor of a custom middleware class but allows the addition of dependencies with parameters through the `Invoke` and `InvokeAsync` methods.

Obtaining HTTP context objects in middleware

As an alternative to using DI in the `InvokeAsync` method, it is possible to use the `context.RequestService` property (`https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.httpcontext.requestservices?view=aspnetcore-9.0`), as shown in the following code:

```
public async Task InvokeAsync(HttpContext context)
{
  var logger = context.RequestServices
    .GetRequiredService<Ilogger
    < BeforeAfterRequestMiddleware >>();
  logger.LogInformation($"Request:{context.Request.Method}
    {context.Request.Path}");
  await _next(context);
  logger.LogInformation($"Response:
    {context.Response.StatusCode}");
}
```

However, this somewhat decreases dependency visibility in code.

However, ASP.NET Core 9 offers an approach to enabling the use of custom middleware on a per-request basis using factory-based middleware, which we will discuss in the next section.

Working with factory-based middleware

Another way to create custom middleware is by using the factory-based approach, which offers better performance and flexibility using DI.

This approach is particularly useful when the middleware requires scoped services.

Factory-based middleware uses the `IMiddleware` interface, which allows the middleware to be activated by the **DI container** (**DIC**).

The `IMiddleware` interface has only one `InvokeAsync` method that must be implemented in the class. The structure of a customized middleware that uses the factory-based approach is very similar to the traditional approach learned in the previous section.