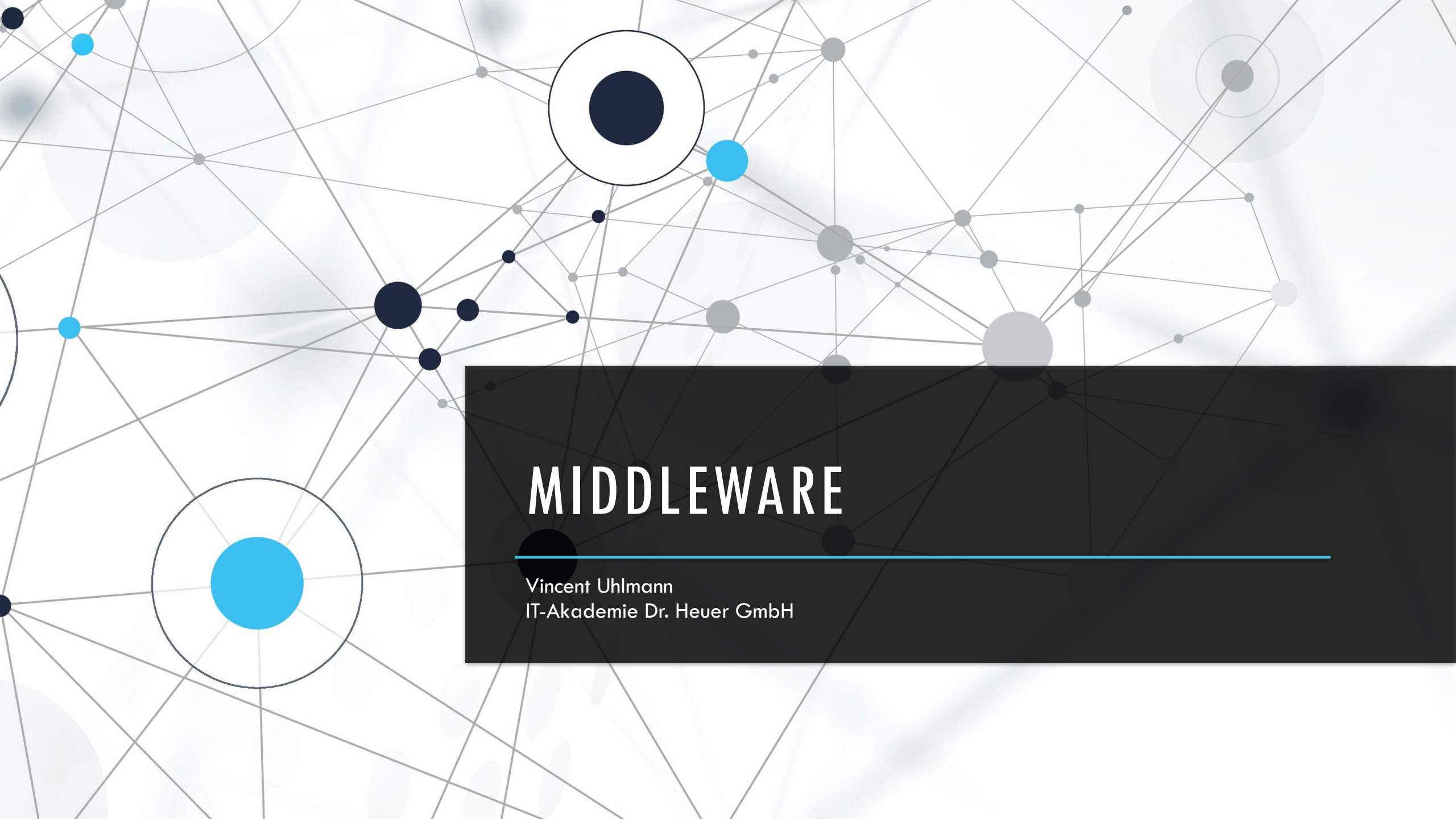


MIDDLEWARE

Vincent Uhlmann
IT-Akademie Dr. Heuer GmbH

MIDDLEWARE IN ASP.NET CORE

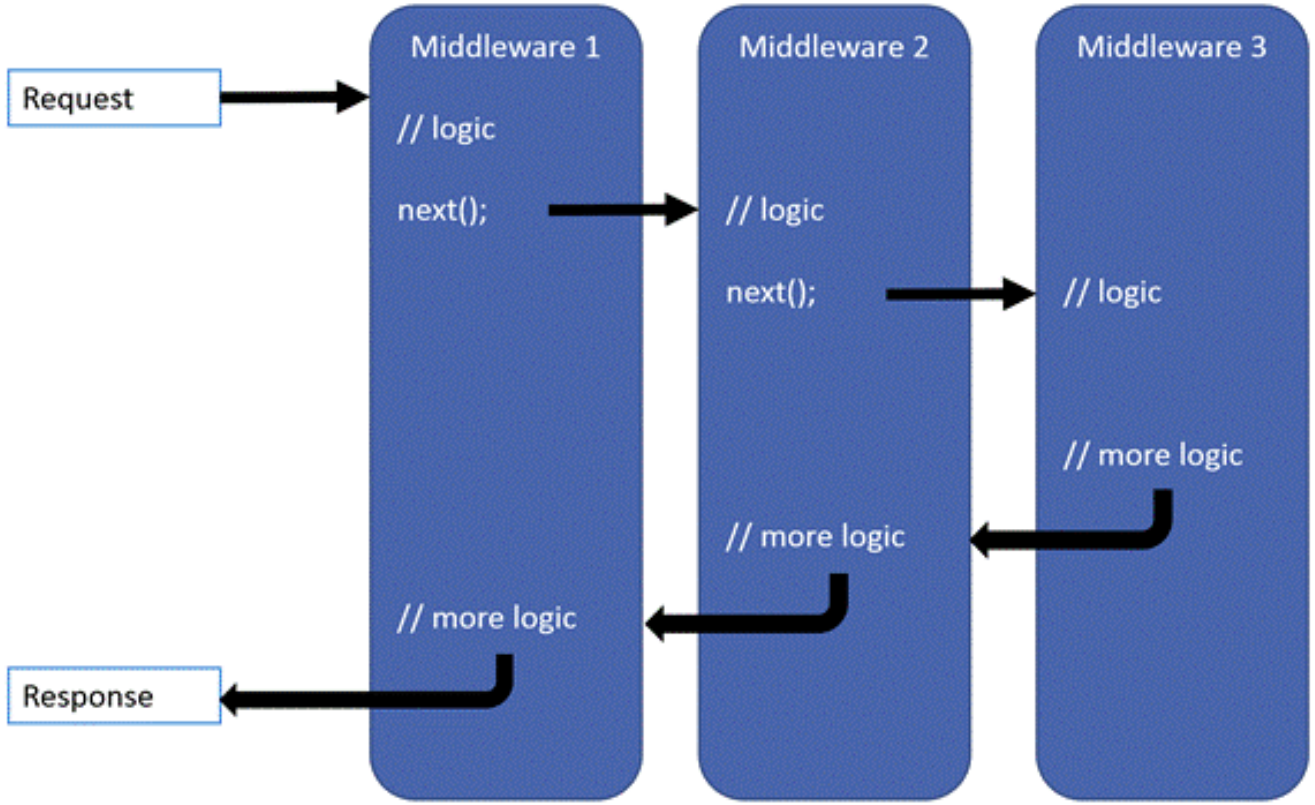
- Middleware sind Softwarekomponenten, die eine Anwendungs-Pipeline definieren und HTTP-Anfragen und -Antworten verarbeiten
- Ermöglicht die Konfiguration der Anwendungs-Pipeline zur Handhabung von HTTP-Anfragen und –Antworten
- Middleware spielt eine entscheidende Rolle bei der Verarbeitung von HTTP-Anfragen in Web APIs
- Middleware wird verwendet, um spezifische Logik einzubinden, die auf jede Anfrage angewendet werden soll

MIDDLEWARE IN ASP.NET CORE

- **Funktion**
- Middleware-Komponenten entscheiden, wie Anfragen behandelt werden und wie Antworten an den Client zurückgesendet werden
- Jede Middleware-Komponente kann Anfragen verarbeiten und die Kontrolle an die nächste Komponente weitergeben
- Bietet Flexibilität und Anpassungsfähigkeit, um spezifische Anforderungen zu erfüllen

- **Beispiele**
- Authentifizierung, Logging, Caching, Routing…

REQUEST PIPELINE



MIDDLEWARE METHODEN

- **Use**
- Fügt Middleware zur Pipeline hinzu, die die nächste Middleware aufruft

- **Run**
- Fügt Middleware hinzu, die die Verarbeitung der Pipeline beendet (z.B. etwas ausführt)

- **Map**
- Teilt die Pipeline basierend auf einer Bedingung, typischerweise dem URL-Pfad

INLINE MIDDLEWARE

- Inline Middleware wird direkt in der Program.cs Klasse definiert
- Next steht hier für den nächsten Delegaten in der Pipeline
- Wenn next nicht aufgerufen wird, wird die Pipeline unterbrochen

```
app.Use(async (HttpContext context, RequestDelegate next) => {
    // Logik vor dem nächsten Middleware-Aufruf
    await next.Invoke();
    // Logik nach dem nächsten Middleware-Aufruf
});
```

MIDDLEWARE (KLASSE)

- Bei einer Middleware Klasse, wird eine Klasse wie z.B. FooMiddleware angelegt die folgendes enthalten muss

- Einen öffentlichen Konstruktor mit einem Parameter des Typs RequestDelegate

- Eine öffentliche Methode mit dem Namen Invoke oder InvokeAsync
- Diese Methode muss einen Task zurückgeben
- Einen ersten Parameter des Typs HttpContext akzeptieren
- Weitere Parameter für den Konstruktor und Invoke/InvokeAsync werden mittels Dependency Injection aufgelöst

MIDDLEWARE (KLASSE)

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

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

    public async Task Invoke(HttpContext context)
    {
        await context.Response.WriteAsync("Hello from middleware! 1");
        await _next(context);
        await context.Response.WriteAsync("Hello from middleware! 2");
    }
}
```

MIDDLEWARE (KLASSE)

- Um die Middleware zu registrieren, muss diese in der Program.cs hinzugefügt werden

```
app.UseMiddleware<FooMiddleware>();

app.Run();
```

- Zusätzliche Konstruktorparameter können angehängt werden

```
app.UseMiddleware<RequiredHeaderMiddleware>("Authorization", "SuperSecretKey");
```

BEARBEITEN DER RESPONSE

- Header können hinzugefügt oder geändert werden, bevor die Antwort an den Client gesendet wird
- Dasselbe gilt für den Status Code, auch dieser kann nur geändert werden, bevor die Antwort an den Client gesendet wird

```
app.Use(async (httpContext, next) => {
    httpContext.Response.Headers.TryAdd("X-Custom-Header", "CustomHeaderValue");
    await next.Invoke();
});
```

BEARBEITEN DER RESPONSE

- Um zu überprüfen ob die Antwort bereits gestartet wurde, bevor Header oder Statuscode geändert werden, kann HasStarted genutzt werden

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

    if (!httpContext.Response.HasStarted)
    {
        httpContext.Response.StatusCode = StatusCodes.Status200OK;
        httpContext.Response.Headers.TryAdd("X-Post-Processing", "AfterStart");
    }
});
```

ABBRECHEN DER REQUEST PIPELINE

- Wenn z.B. Falsche Header in einer Anfrage vorhanden sind, kann die Pipeline abgebrochen und eine Antwort zurückgegeben werden

```
app.Use(async (context, next) =>
{
    if (!context.Request.Headers.ContainsKey("X-Required-Header"))
    {
        context.Response.StatusCode = StatusCodes.Status400BadRequest;
        await context.Response.WriteAsync("Bad Request: Missing X-Required-Header.");
        return;
    }

    await next();
});
```

REIHENFOLGE DER MIDDLEWARE

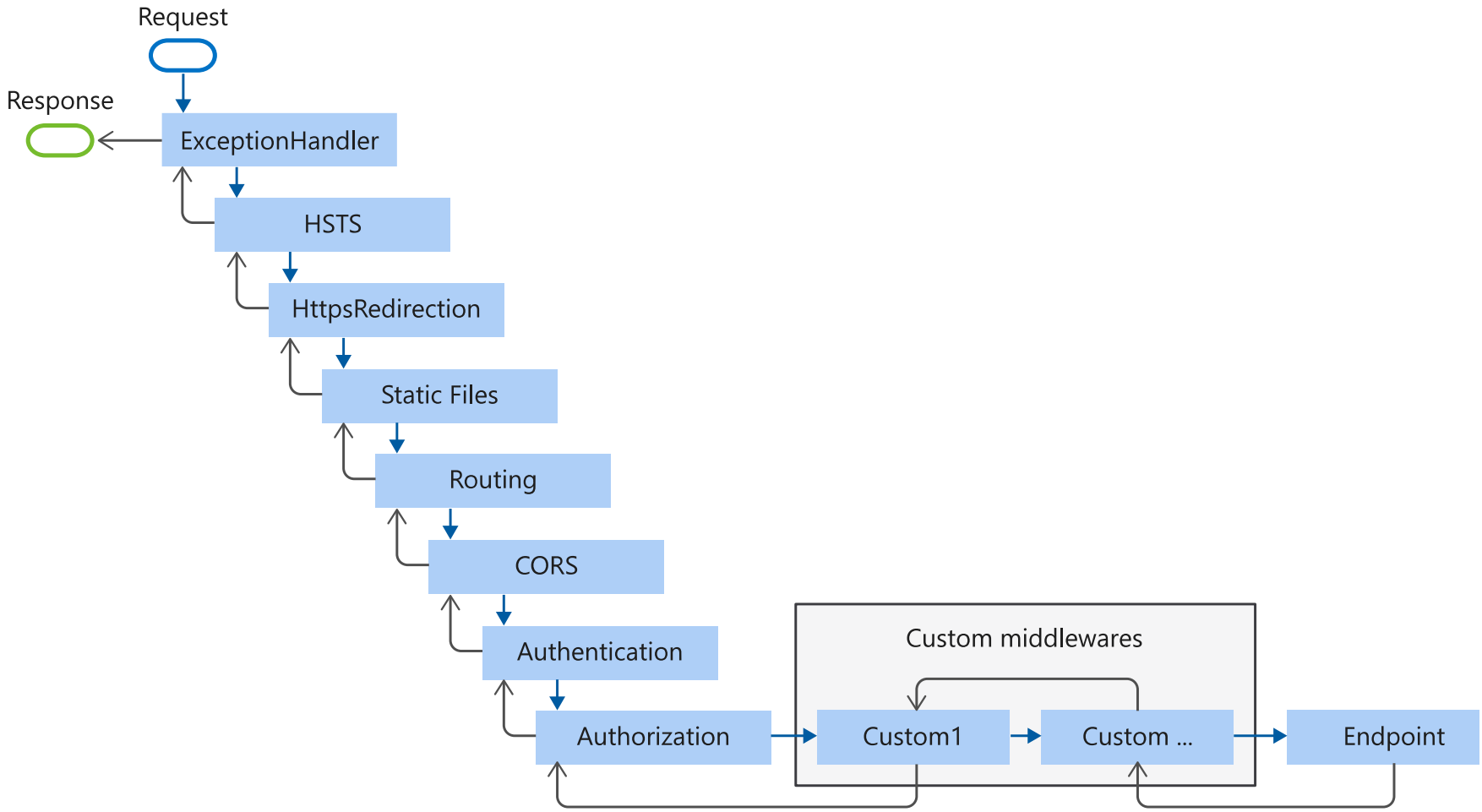
- Die Reihenfolge der Middleware-Aufrufe ist **entscheidend** für das Verhalten der Anwendung
- Jede Middleware-Komponente ruft die nächste Komponente auf oder beendet die Verarbeitung
- Authentifizierungsmiddleware sollte vor Autorisierungsmiddleware aufgerufen werden, um sicherzustellen, dass nur authentifizierte Benutzer autorisiert werden

```
app.UseMiddleware<FooMiddleware>();
app.UseMiddleware<TestMiddleware>();

// Ist NICHT das gleiche wie

app.UseMiddleware<TestMiddleware>();
app.UseMiddleware<FooMiddleware>();
```

REIHENFOLGE DER MIDDLEWARE



TYPEN VON MIDDLEWARE

- ASP.NET Core kommt mit einer Reihe von integrierter Middleware

- **Beispiele**
- UseAuthentication(): Verarbeitet Authentifizierungsanforderungen
- UseAuthorization(): Handhabt Autorisierungsregeln
- UseCors(): Konfiguriert Cross-Origin Resource Sharing (CORS) Einstellungen
- UseEndpoints(): Definiert Endpunkte für Anfragen (z.B. MVC, Razor Pages)
- UseStaticFiles(): Dient zum Bereitstellen statischer Dateien (z.B. HTML, CSS, JS)