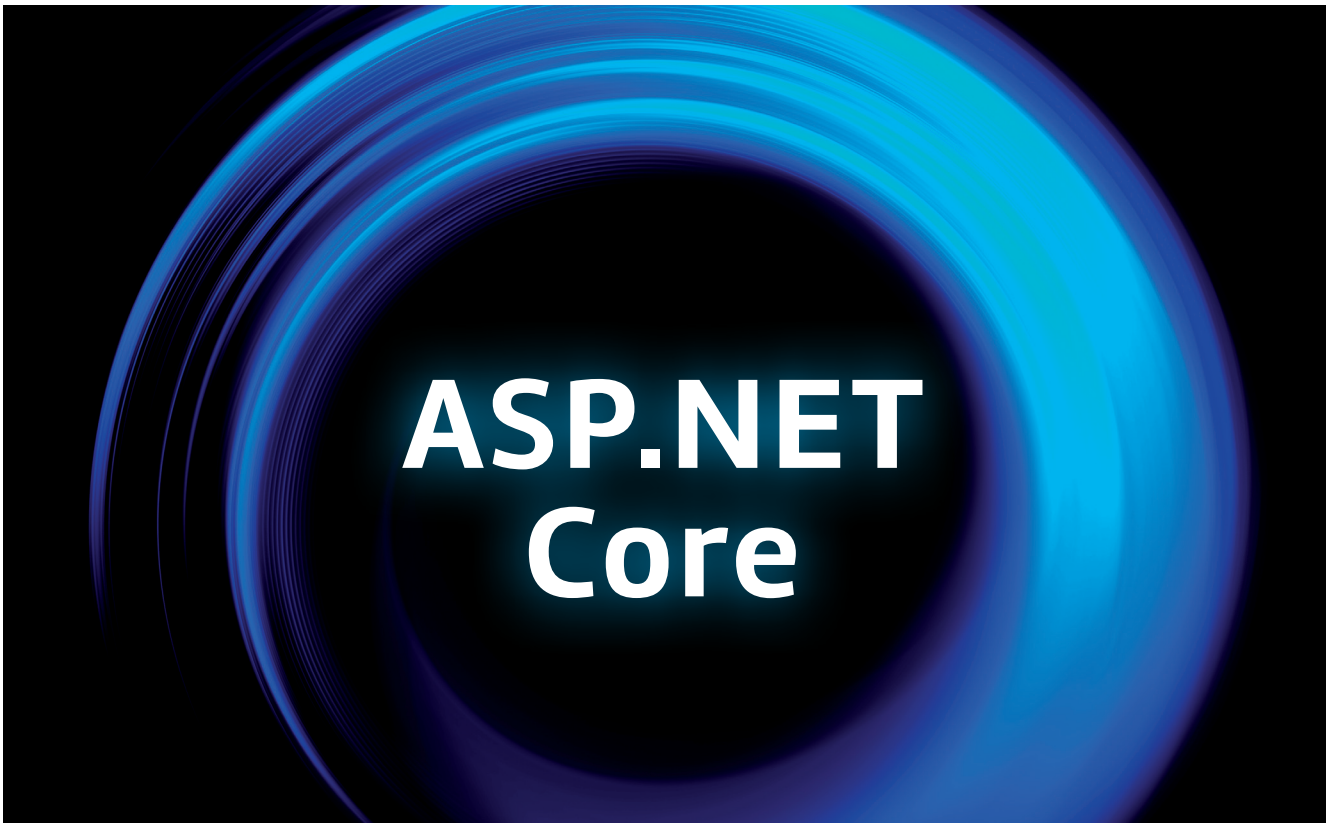


DIE ASP.NET CORE MIDDLEWARE

Die eigene Pipeline

Das Middleware-Pattern und die dazugehörigen Ansätze.

Die ASP.NET-Middleware ist eines der zentralen Elemente in jeder ASP.NET-Anwendung. Versteht man ihren Aufbau und weiß, wie man diese zu einer Pipeline zusammenstellt, kann man mächtige ASP.NET-Antwortmuster aufbauen. Zusätzlich schult das Entwickeln einer eigenen Pipeline das Verständnis und ermöglicht es, effizienter zu debuggen.

ASP.NET verarbeitet jeden Request in einer Pipeline. Die Pipeline besteht aus mehreren Komponenten, die jeweils eine Aufgabe bei der Verarbeitung des Requests erfüllen. Die Komponenten werden Middlewares genannt. Je nach Anforderung an die Server-Anwendung kann die Pipeline dank des Interceptor-Entwurfsmusters [1] aus standardisierten Middleware-Komponenten zusammengestellt werden.

ASP.NET selbst, aber auch viele Drittanbieter bieten reichlich vorgefertigte Middleware-Lösungen an. Für die meisten Anforderungen gibt es bereits vorgefertigte Middlewares, welche lediglich der Pipeline hinzugefügt werden müssen. Ein geschickter Aufbau der Pipeline erlaubt unter anderem:

- replizierte Logiken in Controllern zu reduzieren,
- sauberes Logging einzubauen,
- unterschiedliche Anforderungen auf einem Server zu erfüllen (MVC, SignalR, Razor, Web API, gRPC),
- Authentifizierung, Autorisierung und Routing klar zu modellieren.

Dieser Artikel zeigt, wie Pipelines aufgebaut werden und wie man eine eigene Middleware erstellt. Die Reihenfolge, in der die Middleware-Komponenten in der Pipeline geschaltet werden, ist in der Regel nicht beliebig! Bei der Integration einer Middleware ist es absolut essenziell, diese an der richtigen Stelle in der Pipeline einzuhängen. Ansonsten kann es zu Bugs in der Bearbeitungskette kommen.

Vom Request zur Response

Jede Middleware in der Pipeline ist nach dem gleichen Schema aufgebaut: Präprozess – Weitergabe des Kontrollflusses an die nächste Middleware – Postprozess, siehe Bild 1.

Durch diese Vorgehensweise kann jede Middleware den Request im Präprozess manipulieren. Typische Fälle sind das Parsen des HTTP-Headers, das Ablegen der Anfrage im *HttpContext* oder das Verarbeiten der Authentifizierung beziehungsweise Autorisierung. Anschließend kann die Verarbeitung an die nächste Middleware weitergegeben werden. Dies geschieht durch Aufrufen der *next*-Methode. Dazu übergibt

ASP.NET jeder Middleware als Input einen Zeiger zur nächsten Middleware.

Nachdem die *next*-Methode zurückgekehrt ist, lässt sich die Response im Postprozess ebenfalls manipulieren. Zudem ist es im Postprozess möglich, zu prüfen, ob es zu einer Exception gekommen ist oder ob die Antwort von einem Endpunkt erfolgreich bearbeitet werden konnte. Dadurch lassen sich beispielsweise Fehlerseiten und Performance-Logging einbauen.

Wichtig: Die Response sollte niemals verändert werden, nachdem der Antwortvorgang zum Client begonnen hat. Das kann zu einer Protokollverletzung führen (HTTP).

Eine Middleware, die keine *next*-Methode aufruft, nennt man Terminal-Middleware. Eine Terminal-Middleware ist somit die letzte Middleware in der Kette, die aufgerufen wird. Nach der Terminal-Middleware werden die Postprozesse der vorher besuchten Middlewares in umgekehrter Reihenfolge bearbeitet. Eine Middleware kann auch abhängig vom Request eine Middleware oder Terminal-Middleware sein. Ein Beispiel dafür ist die Endpoints-Middleware (siehe Routing mit *UseRouting* und *UseEndpoints*), die nur dann eine Terminal-Middleware ist, wenn sie die Route verarbeiten kann.

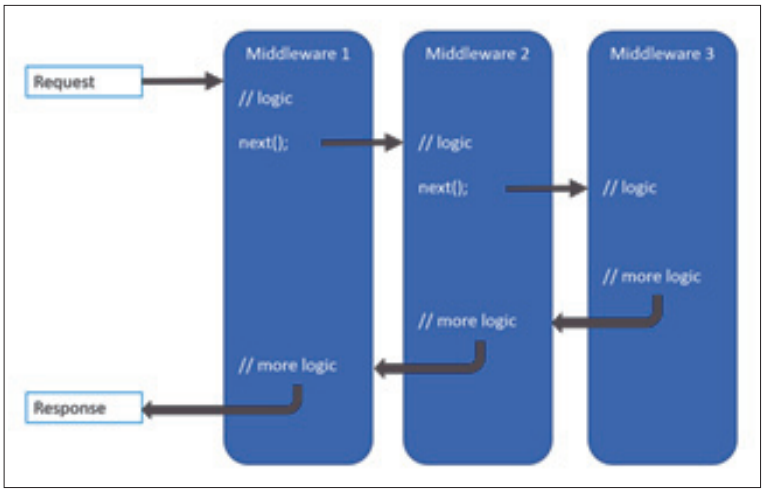


Bild: Microsoft.com

Schematischer Aufbau einer Middleware Pipeline (Bild 1)

Wo befindet sich die Middleware?

Erstellt man ein neues Core-Web-App-Projekt unter ASP.NET 6 mit der Einstellung „Do not use top-level statements“, befindet sich der Aufbau der Pipeline unterhalb der Registrierung der Services in der Datei Program.cs. In älteren .NET-Versionen gibt es eine separate Startup.cs-Datei, die den Aufbau der Anforderungs-Middleware beschreibt.

Alle Methoden, die dem Namensschema *app.UseXYZ()* entsprechen, registrieren eine Middleware in der Pipeline.

Da die Registrierung der Services in der *ServiceCollection* nichts mit dem Aufbau der Middleware zu tun hat, empfiehlt es sich, die beiden Bereiche (Registrierung der Services und Aufbau der Middleware) nach dem Single Responsibility Principle in unterschiedliche Methoden zu gliedern.

Darüber hinaus spielt bei der Middleware die Reihenfolge eine wichtige Rolle; anders ist das bei der Registrierung in der Service-Collection: Hier spielt die Reihenfolge keine Rolle.

Typische Pipeline und verbreitete Middlewares

ASP.NET sowie Drittanbieter stellen diverse Middlewares zur Verfügung. Nachfolgend wird eine Handvoll weit verbreiteter Middlewares aufgeführt. Informationen zur exakten Verwendung und mehr Details finden Sie in der Dokumentation.

Bild 2 zeigt einen typischen Aufbau einer Pipeline mit diversen Middlewares. Es gibt neben dieser Pipeline noch eine MVC-Filter-Pipeline und eine Razor-Pages-Filter-Pipeline. Diese sind aber außerhalb des Scopes des Artikels und spezielle Teile von MVC beziehungsweise Razor. Verzichtet wird hier auch auf eine detaillierte Betrachtung des Authentication- und Authorization-Subsystems, da diese den Rahmen des Artikels sprengen würde.

Exceptionhandler [2] prüft, ob unbehandelte Ausnahmen in der Pipeline aufgetreten sind. Falls ja, kann eine Fehlerseite angezeigt werden. Dies ist gerade in Produktionsszenarien sinnvoll, da sonst sensitive Informationen offengelegt werden können, wie beispielsweise der Stacktrace zur Exception, der ein Sicherheitsrisiko sein kann. Diese Middleware sollte möglichst weit am Anfang der Pipeline stehen, damit sie auch jede Exception mitbekommt.

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

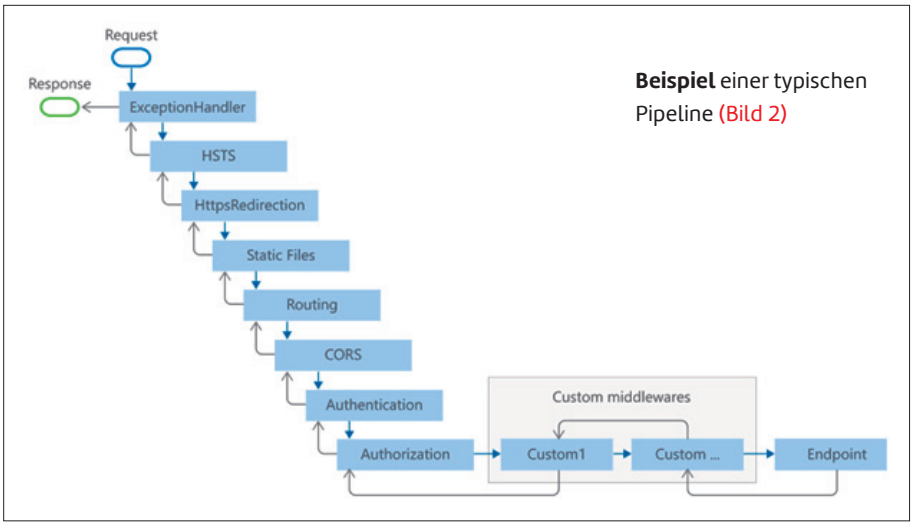


Bild: Microsoft.com

Beispiel einer typischen Pipeline (Bild 2)

HSTS [3] implementiert das HTTP Strict Transport Security Protocol. HSTS ist ein Browser-Sicherheitsfeature, welches verhindert, dass nicht vertrauenswürdige oder ungültige Zertifikate verwendet werden, und welches erzwingt, dass die komplette nachfolgende Kommunikation über HTTPs erfolgt. Da es sich um ein Browser-Feature handelt, hat diese Middleware weniger Sinn bei der Maschine-zu-Maschine-Kommunikation. Es empfiehlt sich, diese Middleware im Development zu deaktivieren, analog zum Exceptionhandler.

HttpsRedirection [4] Falls der Browser eine HTTP-Verbindung zum Server aufbaut, wird ein 307-Temporary-Redirect nebst HTTPS-Adresse zurückgegeben. Diese Middleware ist nur dann sinnvoll, wenn die SSL-Terminierung nicht bereits durch einen ReverseProxy oder Loadbalancer erfolgt.

DefaultFiles [5] gibt Dateien im *wwwroot*-Ordner zurück, ohne dass der Dateiname im Pfad explizit genannt werden muss. Beispiele dafür sind index.html, index.htm, default.html, default.htm. Diese Middleware muss vor StaticFiles platziert sein.

StaticFiles [6] gibt Dateien zurück, die im *wwwroot*-Verzeichnis liegen. Die Middleware kann in Kombination mit DefaultFiles und FileServer genutzt werden und ist nützlich, um statische Assets wie Bilder, Fonts und Dateien der Formate HTML, CSS und JS zur Verfügung zu stellen.

Routing [7] wird in Verbindung mit UseEndpoints oder MapGet, MapPost et cetera genutzt. Nach der Routing-Middleware können nachfolgende Middlewares auf die Routen-Information über den *HttpContext* zugreifen. Routenabhängige Middlewares müssen nach der Routing-Middleware aufgerufen werden. Authentication und Authorization werden zwischen Routing und Endpoints platziert.

CORS [8] steuert das Verhalten beim Cross-Origin-Resource-Sharing. Dies ist ein Sicherheitsfeature des Browsers. Oft sollen Browser nämlich Server-APIs nur dann ansprechen, wenn der Browser auf einem bestimmten Host ist. Mit CORS lassen sich fremde Hosts freischalten beziehungsweise blockieren. Beim CORS macht der Browser einen Pre-Flight-Request, der prüft, ob ein bestimmter Host freigeschaltet ist. Die Server-Antwort wird in der CORS-Middleware berechnet. Beim Entwickeln von SPAs und beim Deployen von fremdgehosteten Diensten findet diese Middleware Einsatz.

Authentication [9] prüft, wer der aktuelle Nutzer ist. Dies kann über unterschiedliche Schemata erfolgen, wie beispielsweise über Bearer-Token, Cookies oder eigene Ansätze. Zudem lassen sich Multi-Factor-Authentifizierungen implementieren. Die Middleware ist sehr komplex und spannt ein eigenes Subsystem auf, auf das dieser Artikel nicht näher eingeht.

Authorization [10] prüft auf Basis der Nutzerinformationen, die von der Authentication-Middleware bereitgestellt wurden, ob der beglaubigte Nutzer eine bestimmte Aktion ausführen darf. Die Modellierung der Berechtigungen kann beispielsweise auf Rollen- oder Policy-Basis erfolgen. Es können auch eigene Richtlinien programmiert werden. Es ist notwendig, dass die Authentication vorher erfolgt ist. Oftmals ist die Autorisierung vom konkreten Endpunkt abhängig, sodass diese nach dem Routing erfolgen sollte.

Endpoint [11] ist für das Verteilen der Anfragen an verschiedene Services verantwortlich. Anfragen können unter anderem an Controller, RazorPages, SignalR und gRPC weitergeleitet werden. Die Route wird in der Routing-Middleware bereits ermittelt. Diese Middleware ist eine Terminal-Middleware, wenn die Route behandelt werden kann. Die Endpoint-Middleware ist meist am Ende der Pipeline zu finden.

Formale Definition einer Middleware

Nachfolgend wird erläutert, was eine Middleware formal ist und wie man eigene Middlewares baut. Dabei verfolgt der Artikel den Bottom-up-Ansatz. Zunächst geht es darum, wie man mit Low-Level-Funktionen eine Middleware baut und registriert. Danach werden schrittweise Ansätze auf höheren Levels vorgestellt.

Die Definition einer Middleware findet sich im Namespace *Microsoft.AspNetCore.Http*:

```
public delegate Task RequestDelegate(
  HttpContext context);
```

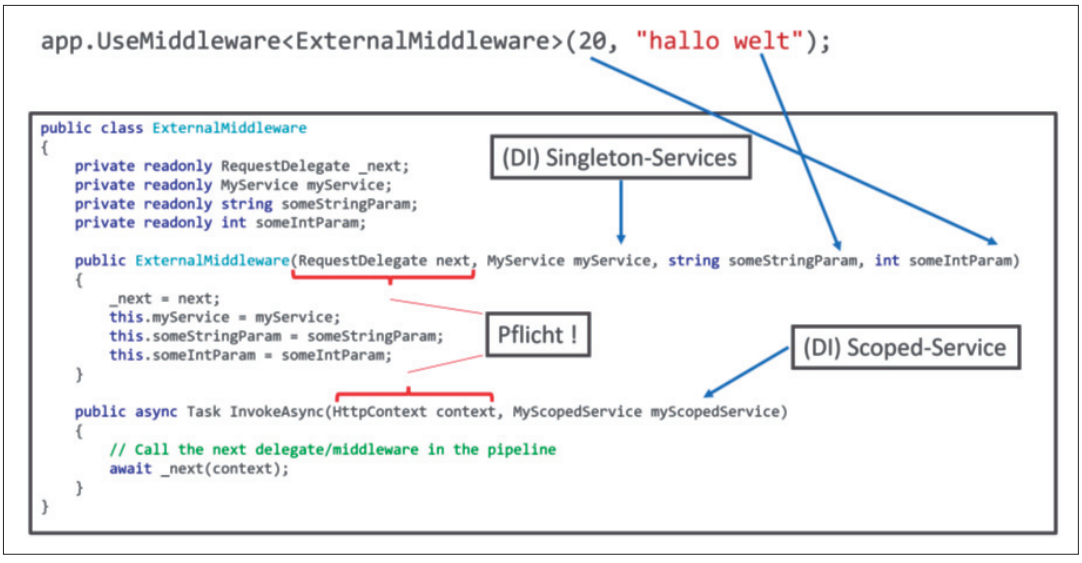
Ein *delegate* ist (wie eine Klasse) ein Typ, der Methodensignaturen definiert. Eine Middleware beziehungsweise ein *RequestDelegate* ist demnach eine Funktion, welche einen *HttpContext* auf einem *Task* abbildet. Der *HttpContext* enthält die zum Request gehörenden Informationen. Einzelne Middlewares können den *HttpContext* auslesen und diesen auch manipulieren.

Inline-Middleware mit dem IApplicationBuilder

Das *IApplicationBuilder*-Interface, das von *WebApplication* erfüllt wird, bietet die direkteste Möglichkeit, um eine Middleware in die Pipeline einzuhängen. Die *Use*-Methode erwartet eine Funktion, die einen *RequestDelegate* konsumiert und einen *RequestDelegate* zurückgibt.

```
/// <summary>
/// Adds a middleware delegate to the application's
/// request pipeline.
/// </summary>
/// <param name="middleware">The middleware delegate.
/// </param>
/// <returns>The <see cref="IApplicationBuilder"/>.
/// </returns>
IApplicationBuilder Use(Func<RequestDelegate,
  RequestDelegate> middleware);

((IApplicationBuilder)app).Use(next =>
{
  RequestDelegate myCustomMiddleware =
    async context => {
      // Präprozess
      await next(context);
      // Postprozess
    };
    return myCustomMiddleware;
});
```



Die Methode UseMiddleware<T>(params) (Bild 3)

In diesem Beispiel wird der *RequestDelegate* namens *next* konsumiert und ein neuer *RequestDelegate* namens *myCustomMiddleware* erstellt und zurückgegeben. Der Parameter *next* verweist auf die nächste Middleware in der Pipeline.

Das Erstellen von *myCustomMiddleware* geschieht nur einmalig beim Programmstart. Dabei werden die Middlewares in umgekehrter Reihenfolge wie die Registrierungen gebaut. Nur so kann ASP.NET den *next*-Parameter zur nachfolgenden Middleware befüllen.

Die *myCustomMiddleware* bekommt zur Ausführungszeit als Parameter den *context*, was den *HttpContext* während der Bearbeitung des HttpRequests darstellt. Der Präprozess, der Aufruf von *next(context)* und der Postprozess werden für jeden HttpRequest durchlaufen. In den Dokumentationen von Microsoft findet sich öfter auch das folgende Konstrukt:

```
((IApplicationBuilder)app).Use(next =>
  async context => {
    // Präprozess
    await next(context);
    // Postprozess
});
```

Diese kürzere Variante folgt derselben Logik, die Unterscheidung, was bei der Erstellung und was bei der Ausführung einer Middleware geschieht, ist allerdings nicht so klar ersichtlich wie in der ausführlicheren Variante.

Inline-Middleware mit der Erweiterungsmethode Use

Neben dem *IApplicationBuilder* stehen, wie im ASP-Bereich üblich, auch Erweiterungsmethoden zur Verfügung, welche die Verwendung vereinfachen. Die Extension-Klasse *UseExtensions* befindet sich im Namespace *Microsoft.AspNetCore.Builder*.

```
// Variante 1
public static IApplicationBuilder Use(
  this IApplicationBuilder app,
  Func<HttpContext, Func<Task>, Task> middleware)

// Variante 2
public static IApplicationBuilder Use(
  this IApplicationBuilder app,
  Func<HttpContext, RequestDelegate, Task> middleware)
```

Die Signaturen der Methoden sehen auf den ersten Blick ziemlich wild aus. Die Verwendung ist aber bei beiden Varianten identisch.

```
// Verwendung Variante 1
app.Use(async (context, next) =>
{
  await next();
});

// Verwendung Variante 2 (empfohlen)
app.Use(async (context, next) =>
{
  await next(context);
});
```

In Variante 1 wird der *context* von der Extension-Methode gecaptured, sodass man den *context*-Parameter nicht übergeben muss. Selbstverständlich lassen sich auch hier die Prä- und Postprozesse integrieren.

Variante 2 wird von Microsoft aus Performance-Gründen empfohlen, weil in Variante 1 der Context immer gecaptured werden muss.

Externe Middleware-Komponenten

Neben der *Use*-Methode und den *Use*-Extensions bietet ASP.NET auch noch die *UseMiddleware*-Extension an, die Sie wie folgt nutzen können:

```
app.UseMiddleware<T>(params);
```

Diese Erweiterungsmethode erlaubt es, eine Middleware in eine eigene Klasse *T* auszulagern. Die folgenden Eigenschaften müssen bei einer externen Middleware eingehalten werden [12]:

- Ein öffentlicher Konstruktor mit einem Parameter des Typs *RequestDelegate*.
- Eine öffentliche Methode mit dem Namen *Invoke* oder *InvokeAsync*. Diese Methode soll einen Task zurückgeben und einen ersten Parameter des Typs *HttpContext* akzeptieren.

Zudem lässt sich Dependency-Injection [13] nutzen, indem man Singleton-Services im Konstruktor oder Scoped- und Transient-Services in den *Invoke*-Methoden übergibt.

Parametrisiert werden kann die Middleware, indem man die Parameter bei *UseMiddleware<T>(params)* übergibt und diese im Konstruktor ausliest (Bild 3).

Auslagern der External-Middleware per Erweiterungsmethode

Die Middleware per *app.UseMiddleware<ExternalMiddleware>* zu nutzen ist gut, aber nicht so konsistent, wie man es von anderen Middlewares kennt. Eigentlich möchte man schreiben: *app.UseMyMiddleware()*. Dies wird möglich, indem man eine Extension-Methode auf den *IApplicationBuilder* aufbaut.

Somit können Middlewares gestaltet werden, die sich nahtlos in den Aufbau der Pipeline integrieren. Außerdem lässt sich so das Open-Close-Prinzip einhalten, indem die Implementierung der Middleware mit einem internen Modifizierer gestaltet wird.

```
public static class MyCustomMiddlewareExtensions
{
  public static void UseMyMiddleware(
    this IApplicationBuilder app)
  {
    app.UseMiddleware<ExternalMiddleware>();
  }
}
```

Routing mit UseRouting und UseEndpoints

ASP.NET kann verschiedene Services betreiben, unter anderem SignalR, Razor Pages, gRPC und einige mehr. Diese lassen sich auf einem einzigen Server in Kombination nutzen. Wichtig ist lediglich, dass man die Routen richtig konfiguriert. Das geschieht in der *UseEndpoints*-Middleware. Sie ist zugleich eine Terminal-Middleware, wenn sie die Route auflösen kann.

```
app.UseEndpoints(configure =>
{
  configure.MapControllers().WithDisplayName(
    "Controller");
  configure.MapRazorPages().WithDisplayName(
    "Razor Pages");
  configure.MapHub<ChatHub>(
    "/chat").WithDisplayName("Chat Hub");
});
```

In diesem Beispiel werden Controller, Razor Pages und ein SignalR-Hub gebunden. Auch verschiedene Display-Namen wurden zugewiesen. Zusätzlich ist es möglich, Metadaten (*MetaData*), einen Namen (*Name*) und einen Gruppennamen (*GroupName*) anzugeben. Diese Informationen können von routingfähiger Middleware ausgelesen werden, sobald die Route mit *UseRouting* aufgelöst wurde.

Das Auflösen der Route geschieht in der *UseRouting*-Middleware. Diese löst die Route abhängig von der Konfiguration in *UseEndpoints* auf. Entsprechend müssen alle routingfähigen Middlewares nach *UseRouting* und vor *UseEndpoints* eingehängt werden.

Um die Route in einer Middleware nach *UseRouting* abzufragen, kann man folgenden Code nutzen:

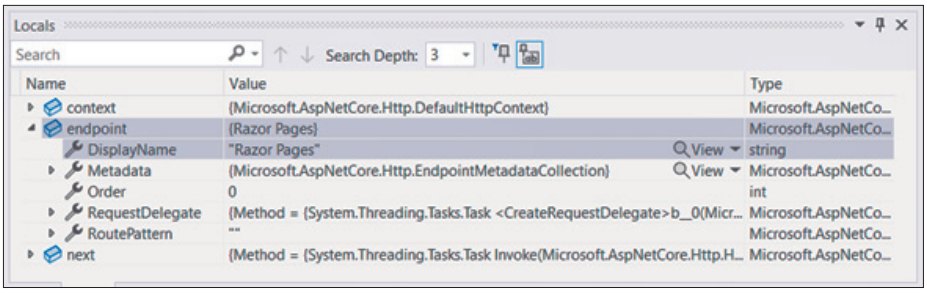
```
var endpoint = context.GetEndpoint();
```

Das *endpoint*-Objekt sieht dann beispielsweise so aus, wie es in Bild 4 gezeigt wird.

Der *DisplayName* ist mit *"Razor Pages"* so gefüllt, wie er in *UseEndpoints* konfiguriert wurde. Die Methode *GetEndpoint* würde *null* zurückliefern, wenn die Methode vor der Routing-Middleware genutzt werden würde.

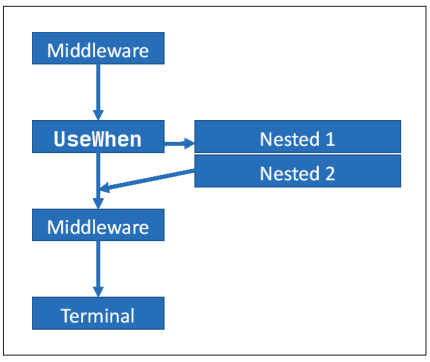
UseWhen, MapWhen und Map

In manchen fortgeschrittenen Fällen ist es erforderlich, bestimmte Middlewares zu überspringen oder sogar eine Abzweigung einzufügen. Dafür gibt es *UseWhen*, *MapWhen* und *Map*. *UseWhen* erlaubt es, eine Middleware nur dann zu verwenden, wenn eine Bedingung erfüllt ist. Die Bedingung wird bei jedem Request erneut ausgewertet. Unabhängig davon, ob die Middleware ausgeführt wird oder nicht, fährt die Pipeline mit derjenigen Middleware fort, die nach *UseWhen* angegeben wurde, vergleiche Bild 5.

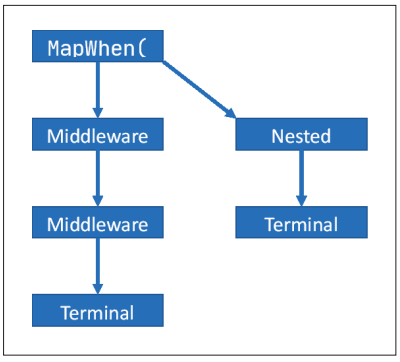


Beispiel für ein Endpoint-Objekt (Bild 4)

```
app.UseWhen(context =>
{
  return context.Request.Method
    == "POST";
```



Ein einfaches Beispiel für den Einsatz von UseWhen (Bild 5)



Terminal-Middleware im Fall einer MapWhen-Abzweigung (Bild 6)

```
}, nestedApp =>
{
  nestedApp.UseMiddleware<MyCustomMiddleware>();
});
```

In diesem Beispiel ist die vorgegebene Bedingung zur Ausführung von *MyCustomMiddleware*, dass es sich bei der Request-Methode um *POST* handelt. Bei einem *GET* oder anderen HTTP-Methoden würde *MyCustomMiddleware* nicht ausgeführt.

MapWhen erlaubt das Abzweigen der Bearbeitung, sofern eine Bedingung erfüllt ist. Diese Methode bekommt analog zu *UseWhen* als ersten Parameter ein Prädikat mit einem *HttpContext*. Wird das Prädikat zu *true* ausgewertet, wird abgezweigt. Im Unterschied zu *UseWhen* erfolgt in diesem Fall kein Aufruf der nachfolgenden Middleware mehr. Wichtig: die Abzweigung bei *MapWhen* und *Map* muss eine Terminal-Middleware besitzen, siehe Bild 6.

Der Unterschied zwischen *Map* und *MapWhen* liegt darin, dass als erster Parameter kein Prädikat mit einer beliebigen Auswertungslogik übergeben wird, sondern ein *string pathMatch*. Entspricht *pathMatch* der Route des Requests, dann wird abgezweigt, ansonsten nicht. *Map* ist also lediglich eine Abkürzung für *MapWhen*, sofern abhängig vom Pfad abgezweigt werden soll. Auch hier gilt, dass die Abzweigung eine Terminal-Middleware benötigt.

Datenaustausch zwischen Middlewares

Manchmal ist es notwendig, Daten zwischen Middlewares auszutauschen. Um dies zu erreichen, gibt es zwei Ansätze: Man kann Scoped Services verwenden oder das Dictionary *context.Items*.

Eine Möglichkeit besteht darin, einen Scoped Service zu registrieren, der die Daten verwaltet, die sich die Middlewares teilen. Durch die Dependency-Injection, welche in *UseMiddleware<T>* bereits integriert ist, lässt sich der Scoped Service anfordern.

Eine weitere Möglichkeit besteht darin, die *Items*-Property des *HttpContext* zu verwenden, welche vom Typ *IDictionary<object, object?>* ist. Um Konflikte mit anderen Middlewares auszuschließen, empfiehlt es sich, als Key *typeof(CustomRequestContext)* zu verwenden:

```
// Setzen der Daten
context.Items[typeof(
  CustomRequestContext)] =
  new CustomRequestContext();

// Lesen der Daten
var fetched = context.Items[typeof(
  CustomRequestContext)] as
  CustomRequestContext;
```

Fazit

In diesem Artikel wurden das Middleware-Pattern und die dazugehörigen Ansätze vorgestellt. Nach einer kurzen Einführung der relevantesten und bekanntesten Middlewares wurde erklärt, wie man mit *IApplicationBuilder* eine eigene Middleware aufbaut. Darüber hinaus gibt es Methoden wie *app.Use* und *app.UseMiddleware<T>*, welche das Erstellen einer Middleware vereinfachen und Dependency-Injection ermöglichen. Des Weiteren zeigte der Artikel, wie Routing- und Endpoint-Middleware miteinander verflochten sind und wie man eine routingfähige Middleware erstellen kann, indem man die Routen-Information abfragt. Erweiterte Abfrage-Pipelines wurden gebaut, indem mittels *UseWhen* Middlewares übersprungen und mittels *MapWhen* beziehungsweise *Map* abgezweigt wurde. ■

[1] *Michael Eichberg (TU Darmstadt), The Interceptor Architectural Pattern, www.dotnetpro.de/SL2211Middleware1*
[2] *ExceptionHandler Middleware, www.dotnetpro.de/SL2211Middleware2*
[3] *HSTS, www.dotnetpro.de/SL2211Middleware3*
[4] *HTTPs Redirection, www.dotnetpro.de/SL2211Middleware4*
[5] *DefaultFiles, www.dotnetpro.de/SL2211Middleware5*
[6] *StaticFiles, www.dotnetpro.de/SL2211Middleware6*
[7] *Routing, www.dotnetpro.de/SL2211Middleware7*
[8] *CORS, www.dotnetpro.de/SL2211Middleware8*
[9] *Authentication, www.dotnetpro.de/SL2211Middleware9*
[10] *Authorization, www.dotnetpro.de/SL2211Middleware10*
[11] *Endpoints, www.dotnetpro.de/SL2211Middleware11*
[12] *Voraussetzungen von UseMiddleware, www.dotnetpro.de/SL2211Middleware12*
[13] *Dependency Injection und Lebensdauer, www.dotnetpro.de/SL2211Middleware13*

Tim Borowski
hat einen Master-Abschluss in Informatik von der TU Darmstadt und ist seit vielen Jahren als Softwareentwickler tätig. Zusätzlich ist er freiberuflicher Berater und Konferenzsprecher zu den Themen Web, .NET und GPGPU.

dnpCode A2211Middleware