NEUES IN ASP.NET CORE 9

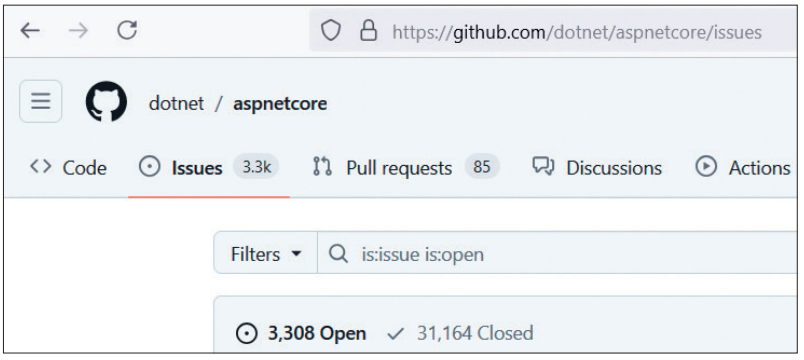
Ausbesserungsarbeiten im Web

Was ändert sich bei der Webentwicklung mit Microsoft?

Wer erfolgsverwöhnt ist, muss mit Enttäuschungen rechnen. Das lässt sich in gewisser Hinsicht bei .NET beobachten, insbesondere bei der Webentwicklung. Galten früher neue .NET-Versionen als sehnlich erwartetes Highlight, ist heute die Begeisterung ein wenig abgeklungen. Salopp formuliert: Was soll denn noch Neues kommen? Der Web-Stack von Microsoft ist – vielleicht mit Ausnahme des jüngsten Mitglieds, Blazor, das in dieser dotnetpro-Ausgabe ab Seite 17 gesondert behandelt wird [1] – etabliert und mehr oder weniger „feature-complete". Das zeigt sich an vielen Stellen: Der MVC-Stack wurde größtenteils noch aus dem „alten" .NET Framework übernommen und dabei nur kosmetisch angepasst, wir reden hier also von einer erstmals 2009 eingeführten Technologie. Razor Pages sind neuer, genauer gesagt aus ASP.NET Core 2, aber das war ebenfalls bereits 2017, und die zentrale Komponente, die Razor-Syntax, stammt aus dem Jahr 2010. Bleibt noch ASP.NET Core Web API. Einst gewissermaßen ein Nebenprodukt von MVC, gab es mit Minimal APIs tatsächlich eine bemerkenswerte Neuerung, aber eher in Richtung Syntax-Zucker. Neuerungen in früheren .NET-Updates schlossen dagegen primär die Feature-Lücke zu Controller-basierten Web-APIs.

Verstehen Sie das bitte nicht falsch, die einleitenden Worte sollen keineswegs zynisch klingen. ASP.NET Core ist einfach in einem so guten Zustand, dass Neuerungen naturgemäß eher kosmetisch ausfallen. Großflächige, nicht abwärtskompatible Änderungen sind von Microsoft weder kapazitätsmäßig zu stemmen noch anhand der zum Redaktionsschluss 3300 offenen GitHub-Issues [2] (Bild 1) sinnvoll priorisierbar. Davon abgesehen würde die Kundschaft sicherlich Sturm laufen, und das mit Recht. Und so fällt das Gros der Änderungen der letzten .NET-Updates meist in eine der folgenden beiden Kategorien:

- **Viele kleinere Neuerungen,** die sinnvolle, aber nicht entscheidende Ergänzungen bieten. Nicht selten findet sehr schnell eine Gewöhnung an die neuen Möglichkeiten statt, sodass es kaum vorstellbar ist, jemals ohne sie gearbeitet zu haben. Dennoch sind es meist Kleinigkeiten. Manche eher signifikante neue Features werden gerne separat entwickelt; ein schönes Beispiel dafür sind die in der vorangegangenen dotnetpro-Ausgabe gezeigten Smart Components [3], die in einem separaten GitHub-Repository – außerhalb der .NET-Organisation! – verfügbar gemacht wurden, zumindest bis jetzt.
- **Verbesserungen unter der Haube,** die die Performance erhöhen: schnellere Ausführung, weniger Speicherverbrauch und dergleichen. Gerne wird das mit Statistiken untermauert. Es ist ein zweischneidiges Schwert, sich damit zu brüsten, denn ein „200 Prozent schneller" (die Zahl ist willkürlich und hat nichts mit ASP.NET Core 9 zu tun) bedeutet ja im Umkehrschluss, dass die Geschwindigkeit in der Vorgängerversion nicht gut war.



Sisyphosarbeit: Ganz schön viele offene Issues bei ASP.NET Core (Bild 1)

In vielen Fällen bedeutet eine neue .NET-Version also rein aus Feature-Sicht: Ein Update kann gemacht werden, muss aber nicht unbedingt erfolgen. Bleibt noch der Support-Aspekt: Geradzahlige .NET-Versionen erhalten drei Jahre Support, oder genauer gesagt: drei Jahre nach Veröffentlichung, und dann bis zum nächsten Patch-Dienstag, auch wenn es derselbe Tag ist. Der zweite Dienstag im Monat ist der Tag, an dem es Updates gibt; es ist also kein Zufall, dass .NET auch in den letzten Jahren immer in der zweiten Woche des Novembers erschienen ist, an einem Dienstag. Bei ungeraden .NET-Versionen läuft es ähnlich, aber der Support endet bereits nach anderthalb Jahren. Für .NET 9 bedeutet das also: Ungerade Versionsnummer, es gibt Support bis Mai 2026. .NET 8 wiederum wird als gerade Version drei Jahre unterstützt, das bedeutet bis November 2026 und damit länger.

Das führt zu einer interessanten Situation: Wenn eine neue .NET-Version kein Feature mit sich bringt, das für ein Projekt

von großem Vorteil oder gar vonnöten ist, sind viele Firmen zögerlich beim Update. In der Praxis kann das insbesondere bei komplexen Softwareprodukten festgestellt werden (Stichwort externe Abhängigkeiten). Der Support-Zeitraum für .NET ist in diesen Fällen das maßgebliche Kriterium. In der Praxis heißt das oft: Alle zwei Jahre wird zur nächsten – geradzahligen – Long-Term-Support-Version migriert, die ungeradzahligen Updates werden übersprungen.

Somit ist für ASP.NET Core 9 ein genauer Blick auf das notwendig, was sich geändert hat. Lohnt sich das Update, oder kann es sinnvoller sein, bis .NET 10 zu warten, das voraussichtlich im November 2025 erscheint und dann bis November 2028 Support genießen würde? Diese Frage lässt sich natürlich nicht allgemein beantworten, zu verschieden sind Anforderungen, Projekte und Kapazität im Entwicklungsteam.

Dieser Artikel soll aber eine Entscheidungshilfe liefern. Ausgehend von der langen Liste der – größtenteils unerheblichen – Neuerungen in ASP.NET Core 9 [4] stellen wir die subjektiv wichtigsten Änderungen vor. Wenn Ihnen gefällt, was Sie sehen, ziehen Sie ein Update in Erwägung. Wenn nicht, sollten Sie natürlich trotzdem eine Migration zur neuen Version prüfen, denn nicht sicherheitsrelevante Verbesserungen sind eher in .NET 9 zu erwarten und nicht im noch unterstützten .NET 8. Andernfalls kann aber auch das Abwarten eine sinnvolle Strategie sein. Nach dem Release von .NET 9 geht es direkt mit den Arbeiten an .NET 10 weiter. Noch ein kurzer Disclaimer zu Beginn: Grundlage dieses Artikels war der Stand von ASP.NET Core 9 zur Preview-Version 7 Mitte August 2024.

SignalR und AOT

Ein Schwerpunkt des neuen Release ist die Arbeit an der Unterstützung für natives AOT für .NET-Inhalte. Bereits mit .NET 7 wurde die Ahead-of-Time-Kompilierung als Option eingeführt, dedizierte AOT-Vorlagen als Teil von .NET inklusive. ASP.NET Core 8 hat für viele Anwendungsfälle die AOT-Unterstützung nachgerüstet, die Kompatibilitätstabelle unter [5] füllt sich sukzessive.

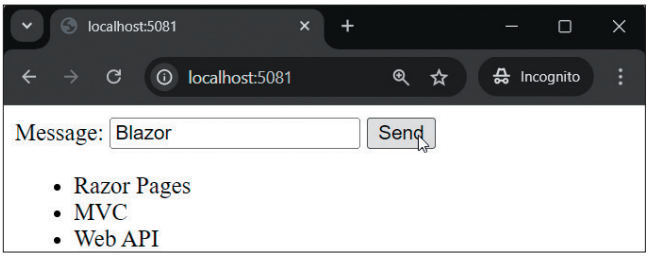
Und auch wenn seither Konsolenanwendungen, Web-APIs und gRPC-Dienste im AOT-Modus laufen können, hatte das in der Praxis zunächst wenig Auswirkungen. Hier lohnt sich ein kurzer Blick auf die Hintergründe. Mit AOT wird eine Anwendung für ein bestimmtes Zielsystem in Form eines Standalonde-Binarys kompiliert. Das Ergebnis läuft dann nicht mehr als Cross-Platform-Anwendung, aber eben auf einer bestimmten Zielarchitektur, und das gut. Microsoft selbst verspricht beim Einsatz von AOT eine schnellere Start-up-Zeit und einen geringeren Speicherfußabdruck [5]. In der Praxis lässt sich das oft, aber nicht immer bestätigen. Eine viel größere Einschränkung ist jedoch, was sich mit AOT ausdrücklich nicht machen lässt: Das dynamische Nachladen von Assemblies und bestimmte Aspekte von Reflection beispielsweise widerstreben dem Ansinnen einer Vorab-Kompilierung. Durch einige Kunstgriffe lassen sich Auswege schaffen, etwa durch die Quellcodegenerierung des JSON-Serialisierers (*System.Text.Json*). Dies ist bereits bei den klassischen Projektvorlagen zu erkennen, hier ein typisches Muster:

```
[JsonSerializable(typeof(string))]
internal partial class AppJsonSerializerContext :
  JsonSerializerContext { }
```

In .NET 9 wurde weiter bei AOT nachgerüstet: SignalR unterstützt jetzt diesen Ausführungsmodus. JSON kommt als Protokoll beim SignalR-Hub zum Einsatz, auf Basis der obenstehenden Klasse:

```
builder.Services.Configure<JsonHubProtocolOptions>(o =>
{
  o.PayloadSerializerOptions.TypeInfoResolverChain
    .Insert(0, AppJsonSerializerContext.Default);
});
```

Eine möglich minimale Hallo-Welt-Anwendung lässt sich auf Basis der Web-API-Projektvorlage (mit „AOT“ im Namen!)



Echokammer: Vorne SignalR-Anwendung ... (Bild 2)

implementieren. Das Template weist nämlich unter anderem zwei Besonderheiten auf, die Sie sonst bei einer bestehenden Nicht-AOT-Anwendung nachrüsten müssten:

- Durch den Einsatz von *WebApplication.CreateSlimBuilder(args)* wird eine „schlanke“ Form der Anwendung erstellt, bei der unter anderem alles hinausfliegt, was mit AOT nicht möglich ist.
- In der Projektdatei sorgt *<PublishAot>true</PublishAot>* dafür, dass auch tatsächlich eine Veröffentlichung als AOT stattfinden kann.

Zur Implementierung: Zunächst definieren wir einen SignalR-Hub:

```
builder.Services.AddSignalR();

// ...

var app = builder.Build();

app.MapHub<EchoHub>("/echo");
```

Der Hub selbst arbeitet mit den Events *Send* und *Receive* und leitet Nachrichten direkt weiter:

```
public class EchoHub : Hub
{
```

▶

```
  public async Task Send(string message)
  {
    await Clients.All.SendAsync("Receive", message);
  }
}
```

Fehlt nur noch das Markup, welches das UI für die Anbindung an SignalR darstellt. Das User Interface besteht aus einem simplen Formular mit Textfeld sowie einer *<ul>*-Liste zur Ausgabe von Werten:

```
<form>
  Message: <input name="message">
  <input type="button" value="Send" onclick="send(
    this.form)">
</form>

<ul id="messages"></ul>
```

Der JavaScript-Code baut zunächst eine Verbindung zum Hub auf:

```
const connection = new signalR.HubConnectionBuilder()
  .withUrl('/echo')
  .build();

connection.start();
```

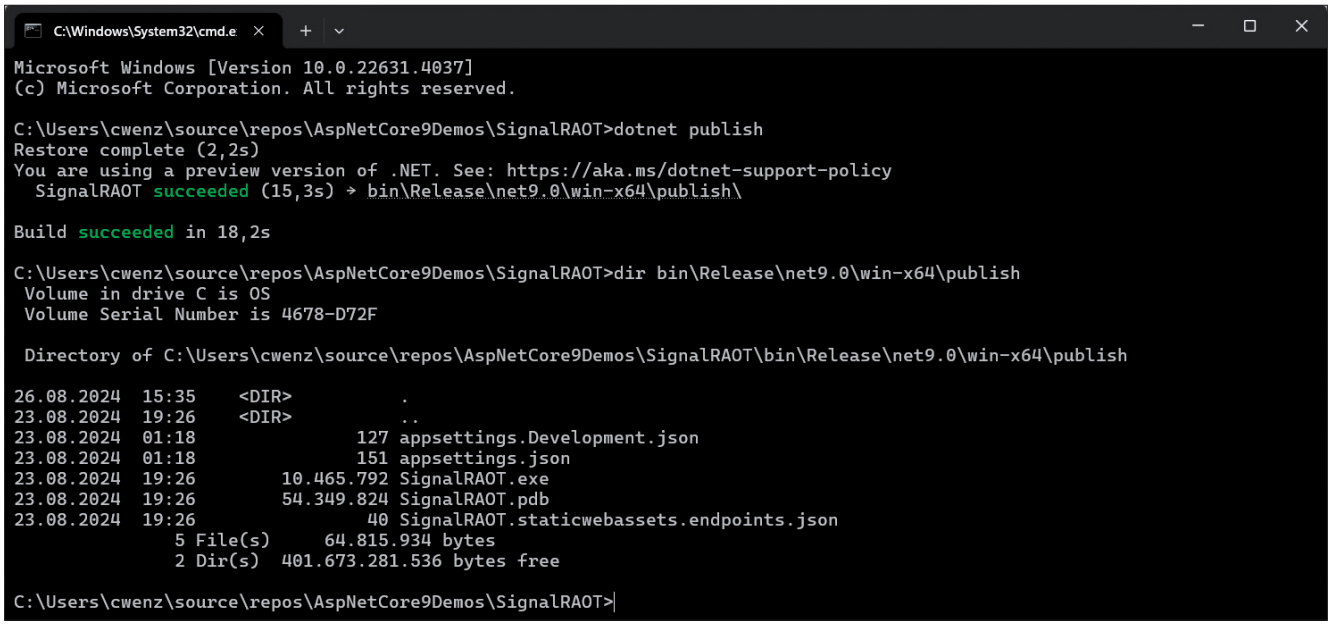
Beim Formularversand wird die Eingabe verschickt …

```
async function send(form) {
  await connection.invoke(
    'Send', form.elements['message'].value);
  form.elements['message'].value = '';
}
```

… wohingegen eingehende Nachrichten direkt in die Oberfläche eingefügt werden:

```
const messages = document.getElementById('messages');

connection.on('Receive', (message) => {
  const li = document.createElement('li');
  li.textContent = message;
  messages.appendChild(li);
});
```



Einzelkämpfer: … hinten AOT (Bild 3)

Bild 2 zeigt das UI. Diese Anwendung können wir mit *dotnet publish* als Stand-alone-AOT-Binary veröffentlichen (in Bild 3 sehen Sie den Prozess und das Ergebnis). Das läuft dann auch tatsächlich nur auf der gewählten Architektur, aber potenziell gut. Bevor Sie begeistert in die Tasten greifen: Prüfen Sie zunächst, ob Ihr Code tatsächlich AOT-kompatibel ist, und falls ja, ob Sie tatsächlich die versprochenen Performance-Vorzüge feststellen können.

Rund um SignalR weist ASP.NET Core 9 noch weitere Neuerungen auf. Die Aktivitätsquelle *Microsoft.AspNetCore.SignalR.Server* löst Ereignisse im Hub aus; zudem können dort jetzt – durch den Einsatz von Basis-Klassen – auch polymorphe Typen verwendet werden.

OpenAPI

Im Lauf des Jahres 2024 hat sich Microsoft-Mitarbeiterin Safia Abdalla auf ihrem X-Account [6] immer häufiger zu den Themen API-Dokumentation, Swagger und OpenAPI geäußert. Besonders letzteres Thema hat es ihr angetan; dabei handelt es sich um eine anerkannte Spezifikation zur Doku-

mentation von HTTP-APIs [7]. Inzwischen ist klar, worauf sie hinauswollte: Bei ASP.NET Core 9 ist der OpenAPI-Support deutlich weiter ausgebaut. Auch in den früheren Versionen war OpenAPI mit dabei, nur der Name nicht immer prominent platziert. Wer jemals mit der Swagger-Oberfläche gearbeitet hat, die bei der Web-API-Projektvorlage automatisch mitgeneriert wird, hat auf OpenAPI gesetzt. Die Checkbox *Enable OpenAPI support* in Visual Studio hat das Projekt entsprechend eingerichtet; bei Verwendung des .NET CLI war gar ein *--no-openapi* notwendig, um die OpenAPI-Unterstützung zu entfernen.

Innenansicht: Das OpenAPI-JSON zum To-do-API (Bild 4)

In ASP.NET Core 9 wurden weitere Features ermöglicht und das NuGet-Paket Microsoft.AspNetCore.OpenApi entsprechend erweitert. Der wohl nützlichste Neuzugang ist die Generierung einer OpenAPI-Dokumentation zu einem implementierten API. Dabei ist es egal, ob Minimal APIs zum Einsatz kommen oder auf Controller gesetzt wird. Zwei Aufrufe werden in der Datei *Program.cs* erwartet:

- *builder.Services.AddOpenApi()* lädt die Middleware,
- *app.MapOpenApi()* verwendet die Middleware und legt die entsprechenden Endpunkte an.

Über den Pfad */openapi/v1.json* erhalten Sie jetzt automatisch eine OpenAPI-Repräsentation der Dienste. Bild 4 zeigt, wie das bei der Standard-Web-API-Vorlage aussieht, wenn AOT zum Einsatz kommen soll. Dort wurde nämlich ein einfaches To-do-API implementiert.

Natürlich ist es nicht sehr effizient, diese JSON-Datei jedes Mal neu zu generieren. Deswegen kann dies auch bereits schon im Build-Prozess geschehen. Dazu muss das Projekt entsprechend konfiguriert werden:

```
<OpenApiGenerateDocuments>true</OpenApiGenerateDocuments>
```

Bei der Generierung der JSON-Definition werden auch in beschränktem Maße Attribute der zugehörigen Modellklassen mitverarbeitet, weil es dafür Entsprechungen in der OpenAPI-Spezifikation gibt. Aus *[DefaultValues]* wird *default*, Modelleigenschaften mit *[Required]* werden in OpenAPI mit *required* markiert.

Für einen weiter reichenden Eingriff in die OpenAPI-Generierung kommen sogenannte Schema-Transformer ins Spiel. Diese definieren eine Blaupause für OpenAPI (Klasse: *OpenApiObject*), die dann exemplarisch in die Datei *v1.json* generiert wird.

Hier ein kleines Beispiel für das To-do-API aus der Web-API-AOT-Vorlage. Microsoft selbst stellt ebenfalls exemplarischen Code zur Verfügung, dieser war allerdings zum Redaktionsschluss so nicht lauffähig, unsere Version hingegen schon:

```
builder.Services.AddOpenApi(options =>
{
  options.AddSchemaTransformer((s
      chema, context, cancellationToken) =>
  {
    schema.Example = new OpenApiObject()
    {
      ["id"] = new OpenApiInteger(42),
      ["title"] = new OpenApiString(
        "Bezeichnug des Todos"),
      ["dueBy"] = new OpenApiDate(DateTime.UtcNow),
      ["isComplete"] = new OpenApiBoolean(false)
    };

    return Task.CompletedTask;
  });
});
```

Das generierte OpenAPI-JSON enthält jetzt wie angegeben ein Beispielelement mit den Angaben aus unserem Code. Tools wie Swagger könnten damit beispielsweise das UI befüllen, um ein Modellelement anzulegen.

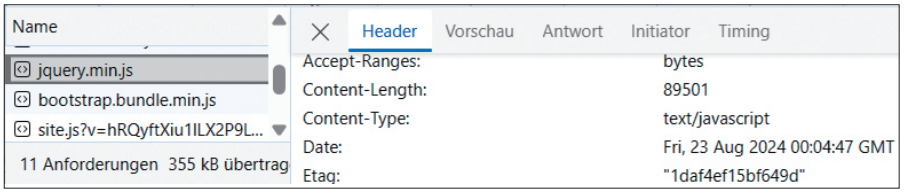
```
"/todos/{id}": {
  "get": {
    "tags": [
      "OpenAPIAOT"
    ],
    "parameters": [
      {
        "name": "id",
```

```
        "in": "path",
        "required": true,
        "schema": {
          "type": "integer",
          "format": "int32",
          "example": {
            "id": 42,
            "title": "Bezeichnug des Todos",
            "dueBy": "2024-08-26",
            "isComplete": false
          }
        }
      }
    ],
    "responses": {
      "200": {
        "description": "OK"
      }
    }
  }
}
```

Statische Assets

Auch in Zeiten von Breitband sind die Datenmenge und die Übertragungsgeschwindigkeit einer Website entscheidende Kriterien und sowohl für ein besseres Google-Ranking als auch für eine höhere Konversionsrate wichtig. Hat gerade jemand laut Blazor WehAssembly gesagt?

Die Techniken, um die Inhalte einer Website möglichst effizient zum Client zu bringen, sind seit Langem bekannt. Unter anderem können Inhalte komprimiert werden, insbesondere HTML und CSS (aber auch Grafiken ohne sichtbaren Qualitätsverlust, doch das ist ein anderes Thema). Zudem ist



Komprimiert: Erste zarte Caching-Ansätze vor ASP.NET Core 9 (Bild 5)

Eine wichtige Änderung an dieser Stelle: Die Swagger-Oberfläche zum direkten Testen des API im Browser wird nicht mehr templateseitig angelegt. Die Begründung des .NET-Teams: Es gebe so viele Alternativen (unter anderem auch den wenig bekannten Endpoints Explorer in Visual Studio), sie möchten nichts Spezifisches vorgeben.

Wenn Sie sich zu sehr an den URL-Endpunkt */swagger* gewöhnt haben sollten: Das NuGet-Paket Swashbuckle.AspNetCore.SwaggerUI bringt Ihnen die alte Funktionalität zurück. Sie müssen allerdings in der Datei *Program.cs* eine entsprechende Konfiguration hinterlegen, damit die Komponente weiß, wo sich die OpenAPI-Definition des API befindet. Den URL haben wir oben schon gesehen; hier der entsprechende Code:

```
app.useSwaggerUI(options => {
  options.SwaggerEndpoint("/openapi/v1.json", "v1");
});
```

Doch kurz zurück zum Anfang der Beschreibung der neuen OpenAPI-Unterstützung. Als Beispiel wurde die AOT-Web-API-Vorlage gewählt, Stichwort To-do-API. Das war kein Versehen, ganz im Gegenteil: Auch die OpenAPI-Generierung ist ein unterstütztes Szenario für natives AOT. Auch hier erstellt *dotnet publish* eine Stand-alone-Anwendung für die gewählte – beziehungsweise, falls nicht angegeben, die aktuelle – Prozessor- und Betriebssystemarchitektur.

Komprimierter: Mehr Caching bei ASP.NET Core 9 (Bild 6)

der Einsatz von Caching empfehlenswert, sodass der Browser beim nächsten Mal bestimmte Ressourcen gar nicht laden muss, sondern sie direkt aus dem Cache bezieht. Allerdings gilt es zu beachten, dass bei Änderungen an den Ressourcen möglichst nicht die zwischengespeicherte Kopie, sondern die aktualisierte Version vom Server verwendet wird.

Mit ASP.NET Core 9 hält eine neue Middleware Einzug, die einige der vorgenannten Techniken einsetzt:

```
app.MapStaticAssets()
```

Sie dient insbesondere als Ersatz für die sonst übliche Middleware für statische Inhalte, *app.UseStaticFiles()*. Dies allerdings nur dann, wenn die Anwendung alle statischen Ressourcen kennt (das schließt insbesondere eingebettete Dateien aus). Die Dateien werden dann zunächst komprimiert, denn Webbrowser unterstützen das automatische Dekomprimieren bestimmter Inhalte schon seit Jahrzehnten. Das verringert zunächst die Datenmenge beim ersten Aufrufen der Anwendung. Dieses Packen von Dateien findet bereits beim Build-Prozess statt, verlangsamt also nur diesen Erstellschritt, aber nicht die Anwendung selbst.

Für noch weniger Bytes beim wiederholten Besuch der Website wird das Caching durch den Browser erleichtert. Die Middleware fügt einen ETag-Header hinzu. Der Wert ist aber nicht zufällig, sondern ist im Wesentlichen der SHA256-Hash des Inhalts der Ressource. Ändert sich diese, ist der ETag-Wert ein anderer und der Browser sollte mit der neuesten Version arbeiten. Gleichzeitig hängt ASP.NET Core 9 mit der neuen Middleware einen Fingerabdruck an den Dateinamen der Ressource an, der automatisch erzeugt wird. Ändert

sich dann beispielsweise eine JavaScript-Datei, würde sich ein neuer Fingerabdruck ergeben und im HTML, das auf den JavaScript-Code verweist, der zugehörige Dateiname. So landet also kein veralteter Code im Browser. Gibt es keine Veränderung, kann der Browser dank Caching-Header mit der bereits vorhandenen Datei arbeiten.

Bild 5 zeigt einige der HTTP-Requests bei Verwendung von *UseStaticFiles()*, Bild 6 die entsprechenden Aufrufe bei Verwendung von *MapStaticAssets()*. Bei der JavaScript-Bibliothek (jQuery) hängt jetzt ein Fingerabdruck am Dateinamen. Die selbst erstellte JavaScript-Datei (*site.js*) trug schon vor der neuen Middleware eine Art von Fingerabdruck im URL; das lag am Einbau mit Tag-Helper *asp-append-version*:

```
<script src="~/js/site.js" asp-append-version=
  "true"></script>
```

Mit *MapStaticAssets()* ist der eindeutige Code nun Teil des Dateinamens.

Etwas irritierend ist noch die Verwendung von zwei ETags (siehe Bild 6). Der kurze entspricht dem, der bereits in alten Versionen von ASP.NET Core zum Einsatz kam; der neue ist wie bereits erwähnt der Hash des Inhalts und damit diejenige Variante, bei der eine Kollision (gleicher Hash) zweier unterschiedlicher Codeversionen quasi ausgeschlossen ist. Möglich, dass sich dies in der finalen Version noch ändert. Der Hauptvorzug des Features liegt darin, dass die Performance-Optimierungen automatisch durch das Framework erledigt werden und so nicht auf dem Webserver konfiguriert werden müssen. Gerade dieser Automatismus kann aber natürlich theoretisch zu Seiteneffekten führen, eben etwa geänderten Dateinamen.

Hybrider Cache

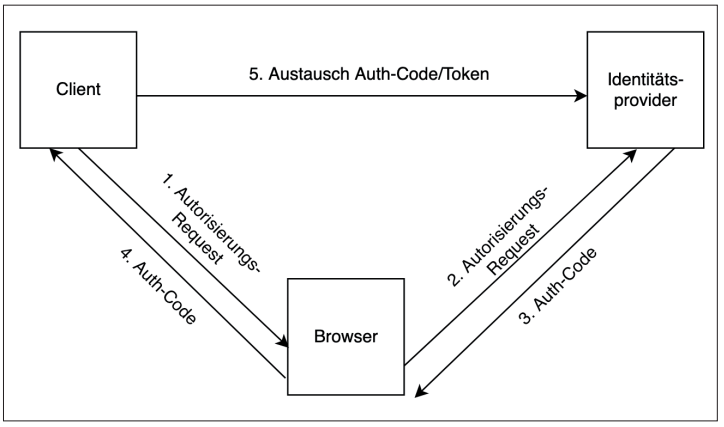
Apropos Caching: Für das Zwischenspeichern im Client haben wir mit *MapStaticAssets()* eine Lösung. Aber auch auf dem Server gibt es hier in ASP.NET Core 9 Neues. Klar, *IDistributedCache* ist ein bestehender, verbreiteter Ansatz (und, in Teilen, auch *IMemoryCache*), allerdings nicht besonders effizient und vor allem ohne Unterstützung für Serialisierung. Im Wesentlichen schreiben wir mühsam Bytes in den Cache und lesen diese Bytes wieder aus. Für 2024 entspricht das nicht der erwarteten Entwicklungs-Experience.

Das neue NuGet-Paket Microsoft.Extensions.Caching.Hybrid bietet eine elegante Alternative. Nach der Registrierung durch *builder.Services.AddHybridCache()* steht *HybridCache* per Dependency Injection zur Verfügung und bietet über die Methode *GetOrCreateAsync()* quasi alles, was benötigt wird: Entweder steht ein Wert schon im Cache zur Verfügung, oder er wird neu erzeugt. Microsoft ist von diesem (überfälligen!) Ansatz so überzeugt, dass er auch in älteren .NET-Versionen eingesetzt werden kann, bis zu .NET Standard 2.0 und sogar .NET Framework 4.7.2 (die NuGet-Seite [8] spricht sogar von 4.6.2, aber wir wollen es nicht gleich übertreiben)!

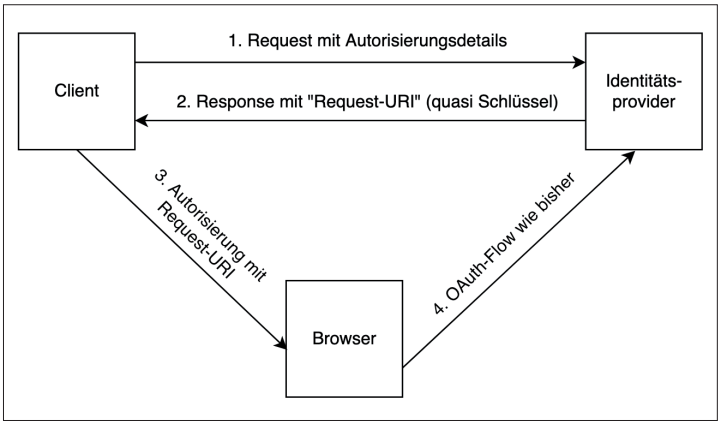
PAR

Auch im Bereich Autorisierung und Authentifizierung gibt es bei ASP.NET Core etwas Neues, nämlich die Unterstützung von PAR. Interessanter als das Feature selbst ist jedoch die Geschichte dahinter.

Zunächst zur Theorie: PAR steht für OAuth 2.0 Pushed Authorization Requests. Das hat von der IETF (Internet Engineering Task Force) die RFC-Nummer 9126 zugewiesen bekommen [9]. Hiermit soll einer der größten Nachteile vieler OAuth- und OIDC-Flows beseitigt werden, nämlich die Weitergabe von Informationen über URLs. Den üblichen groben Ablauf bei Einsatz eines Identitätsproviders zeigt Bild 7 (hier anhand des Flows „Authorization Code with Proof Key for Code Exchange“ in vereinfachter Darstellung): Der Browser möchte auf eine geschützte Ressource zugreifen, wird zum Identitätsprovider weitergeleitet, und von dort geht es zurück zum Client. Je nach Fluss werden aber bei diesem „Rücksprung“ Parameter mitgegeben – etwa der Authentifizierungscode, oder ein Token, oder Scopes, oder weitere Informationen. Es gibt diverse Szenarien, bei denen diese Informationen abgefangen werden könnten (etwa: Cross-Site Scripting), was die Sicherheit des Mechanismus untergraben würde. Abhängig von den beteiligten Systemen haben ▶



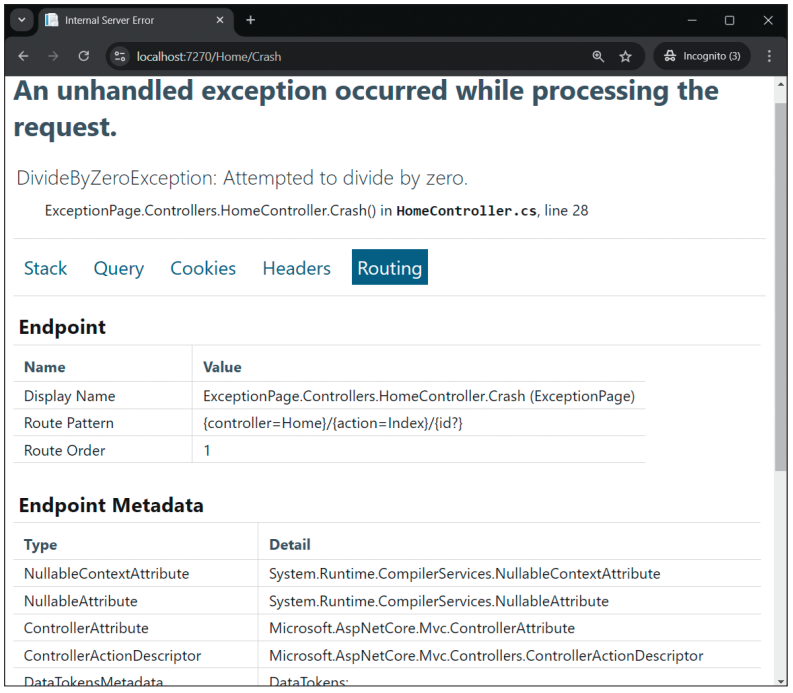
Vornerum: Typischer Ablauf bei einem Identitätsprovider (Bild 7)



Hintenrum: Bei PAR tauschen die Server direkt Informationen aus (Bild 8)

URLs eine Maximallänge, sodass es ein Limit bezüglich der Informationen gibt, die so mitgegeben werden können.

Beim Einsatz von PAR läuft es in etwa wie in Bild 8. Die zuvor über den Client – gerne „Front-Channel" genannt – ausgetauschten Informationen werden jetzt im „Back-Channel" transferiert. Ein clientseitiges Abfangen ist dann nicht mehr möglich, und auch ein durch die URL-Größe vorgegebenes Limit existiert dann nicht mehr. Viele Identitätsprovider unterstützen PAR bereits – und mit ASP.NET Core 9 auch .NET, es ist sogar standardmäßig aktiviert.



Mehr Spaß beim Entwickeln: Die aktualisierte Fehlerseite (Bild 9)

Die eigentliche Überraschung ist die Quelle des Codes: Es ist ein Community-Beitrag, und zwar von dem Team des populären Identitätsproviders IdentityServer [10] (genauer gesagt: einer von dessen Entwicklern, Joe DeCock). Geradezu rührend, wenn man bedenkt wie vergleichsweise wenig Unterstützung IdentityServer selbst als Open-Source-Projekt erhalten hat, was dann schließlich zur Umwandlung in ein kommerzielles Projekt unter dem Dach von Duende Software geführt hat. Diese kleine Firma unterstützt jetzt den großen Redmonder Konzern bei ihren Open-Source-Bemühungen.

Sehr lesenswert ist die Diskussion rund um den API-Vorschlag von DeCock, der dann letztendlich zur Implementierung geführt hat [11].

Dev-Fehlerseite und mehr

Der letzte Punkt, der an dieser Stelle thematisiert werden soll, ist die Verbesserung aus Entwicklungssicht, auch wenn es sich nur um Kleinigkeiten handelt.

Die allseits beliebige Fehlerseite bei nicht abgefangenen Exceptions (die dann auch keinesfalls auf dem Produktivsystem ausgegeben werden sollte) wurde um ein paar wenige Features erweitert. Beispielsweise werden im Bereich „Routing" auch Endpunkt-Metadaten mit aufgeführt (Bild 9), und das Schriftbild (Stichwort Umbrüche) wurde verbessert.

Auch in Visual Studio gab es Verschönerungen, nämlich eine neue Ansicht beim Debuggen von Dictionaries, die jetzt wie eine Liste von Name-Wert-Paaren angezeigt werden. Wieso das als Feature von ASP.NET Core 9 gilt? Das hat sich der Autor dieser Zeilen auch gefragt, aber Microsoft liefert als Begründung, dass es im Web-Stack von .NET viele solcher Dictionaries gibt, beispielsweise bei allem rund um HTTP (Header, Cookies, *GET*- und *POST*-Daten) sowie auch Routen-Infos, um nur einige zu nennen. Das lassen wir jetzt mal als Begründung so stehen. Hieran sieht man schon: Zwar gibt es viele kleinere Neuerungen, aber eine große Linie oder Vision lässt sich daraus nicht ablesen. Möglicherweise gibt es das dann für .NET 10 wieder, dann auch mit drei Jahren Long-Term-Support. ■

[1] Holger Schwichtenberg, Politur, dotnetpro 11/2024, Seite 17 ff., www.dotnetpro.de/A2411Blazor

[2] GitHub-Issues zu ASP.NET Core – Stand 26.08.2024 waren es 3308, www.dotnetpro.de/SL2411ASPNETCore1

[3] Christian Wenz, Kluge Klemmbausteine, dotnetpro 10/2024, Seite 78 ff., www.dotnetpro.de/A2410KIControls

[4] Microsoft Learn, Neuerungen in ASP.NET Core 9, www.dotnetpro.de/SL2411ASPNETCore2

[5] Aktueller Status der Unterstützung von nativem AOT, www.dotnetpro.de/SL2411ASPNETCore3

[6] Safia Abdalla bei X, ehemals Twitter, https://x.com/captainsafia

[7] Homepage der OpenAPI-Initiative, www.openapis.org

[8] Das NuGet-Paket Microsoft.Extensions.Caching.Hybrid, www.dotnetpro.de/SL2411ASPNETCore4

[9] OAuth 2.0 Pushed Authorization Requests, www.dotnetpro.de/SL2411ASPNETCore5

[10] IdentityServer, www.dotnetpro.de/SL2411ASPNETCore6

[11] So kam PAR in ASP.NET Core 9, www.dotnetpro.de/SL2411ASPNETCore7

Christian Wenz
ist Spezialist für die Architektur von Webanwendungen und für Web Application Security. Seit 2004 ist er Microsoft MVP. Seine neuesten Bücher sind „ASP.NET Core Security" (erschienen bei Manning) und „C# und .NET 8" (Hanser).
www.christianwenz.de

dnpCode A2411ASPNETCore