ASP.NET CORE: CACHING

Nachbrenner

Laufzeit- und Antwortverhalten für Webseiten lassen sich mit einigen Kniffen optimieren.

Performance ist in Webapplikationen neben der Sicherheit eines der wichtigsten Themen. Wie auch die Sicherheit wird diese Thematik allerdings meistens nach hinten geschoben und erst später oder gar nicht implementiert. Dabei ist das Laufzeit- und Antwortverhalten für den Kunden sogar fast noch wichtiger als die Sicherheit. Diese Anforderung ist zu Recht in fast jedem Pflichtenheft zu finden.

Allerdings gibt es einen großen Unterschied zwischen Kunden und Entwicklern. Die Anforderung „muss schnell sein" ist für Entwickler ebenso schwer greifbar wie SQL-Injection und Authentifizierung für den Kunden. „Muss schnell sein" kann der Kunde direkt nachvollziehen und fühlen, es ist eventuell sogar messbar. Die schönste Applikation ist dem Kunden nichts wert, wenn er nach jeder Interaktion gefühlt „eine Ewigkeit" warten muss – dabei können drei Sekunden für ihn schon eine solche Ewigkeit sein. Jeder Erklärungsversuch von Entwicklerseite ist danach unnötig, Zeitverschwendung, nicht relevant und nicht nachvollziehbar. Mögen die drei Sekunden auch noch so begründet sein:

- Der Kunde wollte zu viel auf dem UI.
- Die Abfragen sind zu komplex.
- Es gibt zu viele Look-up-Informationen zu laden.
- Die Autorisierung prüft zu viel auf der Datenbank.
- Der Server ist Mist.
- Die Leitung ist zu langsam.

Das sind Erklärungen, die keinen Sinn ergeben. Schließlich sollte die Applikation des Kunden nicht nur schön und funktional sein, sondern auch schnell – was immer das auch heißen mag. Während der Entwickler sich fragt, was „schnell" ist und wie er es messbar und greifbar machen kann, wird sich der Kunde fragen, wo das Problem liegt.

Seien wir aber mal ehrlich und versetzen uns in die Lage des Kunden. Immerhin sind auch Entwickler Kunden und Nutzer verschiedener Entwicklungs-Tools und beschweren sich ständig über deren mieses Laufzeitverhalten. Die genannten Erklärungsversuche sind tatsächlich nichts wert.

Job des Entwicklers ist es, komplexe Dinge einfach zu machen und nicht umgekehrt. Seine Kunden haben ein Problem und er ist da, um es zu lösen. Wenn Datenbankabfragen zu langsam geworden sind, weil die Kundenanforderungen zu komplex sind, muss ein neuer Weg gefunden werden, um das Problem zu lösen und die Abfragen zu beschleunigen.

„Muss schnell sein" ist die einfachste Anforderung, die es gibt. Nicht unbedingt einfach umzusetzen, aber einfach zu verstehen. Eigentlich auch einfach zu prüfen, wenn man mal alle Zahlen und Messungen beiseitelässt. Wenn sich der Nutzer durch die Applikation klickt, sollte er das zügig und ohne lange Wartezeiten machen können.

Listing 1: Cache-Profile anlegen

```
services.AddMvc(options =>
{
  options.CacheProfiles.Add("Any_30",
    new CacheProfile()
    {
    Duration = 30,
      Location = ResponseCacheLocation.Any
  });
  options.CacheProfiles.Add("Client_120",
    new CacheProfile()
    {
      Duration = 120,
      Location = ResponseCacheLocation.Client
  });
});
```

Es gibt eine Reihe von Möglichkeiten, dieses Problem zu lösen: Komplexität auflösen, Abfragen vereinfachen oder in andere Zuständigkeiten verlagern (Datenbankabfragen auf den Datenbankserver verlagern), Code vereinfachen und Daten im Cache vorhalten [1]. Mit Letzterem beschäftigt sich dieser Beitrag.

Caching

Caching ist vom Prinzip her einfach, kann aber schnell zu einem relativ komplexen Problem werden, das gut durchdacht werden sollte. Cachen bedeutet im einfachsten Fall, komplexe Abfragen zwischenzuspeichern, um so die Abfragen zu vereinfachen. Was ist aber eine komplexe Abfrage? Nicht nur die umständliche SQL-Abfrage, sondern eigentlich jeder Zugriff aus der Applikation heraus, sowie Zugriffe auf externe Ressourcen wie Internet, Datenbanken, Netzwerk, große Dateien, Prozesse et cetera – die wichtigsten Stellen, an denen in der Applikation ein Cache implementiert werden sollte.

Eine zweite wichtige Stelle ist das Generieren der Oberfläche des Programms. Diese Aufgabe benötigt Zeit, daher sollte das fertige UI, wenn möglich, ebenfalls gecacht werden.

Auch die dritte wichtige Stelle betrifft den Netzwerkzugriff, diesmal vom Client aus betrachtet. Der Browser sollte auch Ressourcen cachen, die sich nicht oder nicht oft ändern.

Somit sind drei Arten von Caches ausgemacht:

- Browser-Cache: Caching im Client.
- Output-Cache: Caching des gesamten Outputs auf dem Server.
- Application-Cache: Caching spezifischer Daten auf dem Server.

Caching im Client

Der Browser speichert Ressourcen, die sich nicht oft ändern, im Browser-Cache. Das reduziert die Antwortzeiten vom Server erheblich. In den meisten Fällen werden Bilder, Stylesheet- oder auch JavaScript-Dateien auf dem Client zwischengespeichert. Diese Dateien ändern sich zur Laufzeit eher wenig und können vom Browser direkt aus dem Cache geholt werden. Die eigentliche Sicht (View) dagegen kann sich mit den dargestellten Daten immer wieder ändern – je nachdem, was die View macht. Das folgende Codebeispiel zeigt, wie sich das Cache-Verhalten von statischen Dateien anpassen lässt:

```
app.UseStaticFiles(new StaticFileOptions
{
  OnPrepareResponse = context =>
  {
    var response = context.Context.Response;
    response.Headers.Add("Cache-Control",
      "private,max-age=120");
  }
});
```

Das Beispiel zeigt, wie sich mit dem HTTP-Header *Cache-Control* steuern lässt, wie und ob der Client die Dateien zwischenzuspeichern hat. In diesem Fall wird der Client angewiesen zu cachen (*private*), und die Dateien sollen für 120 Sekunden gültig sein. Nach diesem Zeitraum werden die Dateien bei Bedarf neu vom Server angefordert. Alternativ kann statt *private* auch *public* angegeben werden, dann dürfen auch die Proxy-Server cachen. Um das Caching auf den Clients und den Proxys generell zu verbieten, wird *no-cache* angegeben. Die Eigenschaft *max-age* bemisst sich generell nach Sekunden [2].

Bei der Cache-Dauer zeigt sich die erste Schwierigkeit. Hier ist zwischen Aktualität und Performance abzuwägen. Ändern sich die Daten häufig, ist eine kurze Cache-Dauer zu wählen, andernfalls kann sie verlängert werden. Bei den statischen Dateien ist in diesem Fall die Cache-Dauer von zwei Minuten sogar recht kurz. Die Dauer von einem Tag wäre hier sicher angemessen.

Der Browser-Cache kann auch über Attribute in Controller-Actions für einzelne Views gesteuert werden. Somit können nicht nur statische Dateien auf dem Client zwischengespeichert werden, sondern auch dynamische HTML-Seiten.

Über das Attribut *[ResponseCache]* lässt sich die Cache-Dauer (in Sekunden) festlegen: *Location* definiert das Verhalten (*Client* steht für den Browser, *Any* auch für Proxys, *None* bedeutet, nicht zu cachen), und mit der Eigenschaft *NoStore* lässt sich das Caching für diese View generell ausschalten.

Listing 2: Ein einfacher Page-Counter

```
public class PageCounter
{
  private readonly IDictionary<string, long>
    Counters =
    new Dictionary<string, long>();

  public long Count(string counterName)
  {
    if (!Counters.ContainsKey(counterName))
    {
      Counters.Add(counterName, 0);
    }

    var value = Counters[counterName];
    value++;
    Counters[counterName] = value;
    return value;
  }

}

// Registrierung in der Program.cs
services.AddSingleton<PageCounter>();
```

Das folgende Beispiel zeigt das Attribut *[ResponseCache]* mit denselben Werten wie bei *StaticFilesOptions* aus dem obigen Listing:

```
[ResponseCache(
  Location = ResponseCacheLocation.Client,
  Duration = 120,
  NoStore = false)]
```

Dieses Attribut erzeugt den gleichen Antwort-Header, wie er im ersten Beispiel ausgegeben wird.

Als Alternative zu den vielen verschiedenen Angaben in den Attributen lassen sich Cache-Profile anlegen, die über das *[ResponseCache]*-Attribut genutzt werden. Das spart Schreibarbeit und der Code bleibt übersichtlich. Listing 1 zeigt, wie die Profile in der Datei *Startup.cs* angelegt werden. In den Attributen werden dann nur noch die Profile angesprochen:

```
[ResponseCache(CacheProfileName = "Client_120")]
```

Um das zu testen, wird am besten ein neues ASP.NET-Core-MVC-Projekt angelegt. Sowohl in der *Contact*- als auch in der *About*-Aktion (in den Dateien *Contact.cshtml* und *About.cshtml*) wird folgender Code eingefügt:

```
ViewData["Counter"] = __pageCounter.Count("Contact");
ViewData["Time"] = DateTime.Now.ToLongTimeString();
```

Die Werte werden dann auf beiden Views wie folgt ausgegeben:

```
@{
ViewData["Title"] = "About";
}
<h2>@ViewData["Title"]</h2>
<h3>@ViewData["Message"]</h3>

<p>@ViewData["Counter"]</p>
<p>@ViewData["Time"]</p>
```

Nun wird die *Contact*-Aktion mit dem *[ResponseCache]*-Attribut ausgezeichnet. Die *About*-Aktion wird nicht ausgezeichnet, um direkt vergleichen zu können. Navigiert man nun zwischen beiden Views hin und her, nachdem die Applikation gestartet ist, zeigt sich, wie sich der *Counter* und die Uhrzeit in der *About*-View ändern, nicht aber in der *Contact*-View. Je nach Browser führt ein Druck auf die Taste [F5] dazu, dass die Seite vom Server neu geladen wird. Wenn das passiert, ändern sich auch die Werte in der *Contact*-Sicht.

Der hier verwendete *PageCounter* ist eine einfache Klasse, die als Singleton-Instanz im Dependency-Injection-Container gespeichert wird. Listing 2 zeigt dies und die Registrierung im DI-Container.

Output-Caching

Im Gegensatz zum Response-Cache ist beim Output-Cache der Server für das Zwischenspeichern verantwortlich. Aufgrund der Bezeichnung fällt in ASP.NET Core der Output-Cache unter den Response-Cache, obwohl dieser nicht über HTTP-Header die Clients steuert, sondern die Daten direkt auf dem Server speichert.

Der Output-Cache muss aktiviert werden. Dazu muss in der Datei *Startup.cs* in der Methode *ConfigureServices()* die folgende Zeile eingefügt werden:

```
services.AddResponseCaching();
```

In der Methode *Configure()* muss die folgende Zeile hinzugefügt werden:

```
app.UseResponseCaching();
```

Nun sorgt eine Middleware dafür, dass der Output auf dem Server gespeichert wird.

Ob das funktioniert, kann mit den *Vary*-Eigenschaften des Attributs *[ResponseCache]* oder der *CacheProfiles*-Objekte getestet werden. Diese Eigenschaften dienen als Schalter, um eine neue Version des Outputs anzufordern, also um den Cache zu leeren und eine neu erzeugte Sicht zu laden.

Die bisher vorgestellten Eigenschaften funktionieren ohne die *ResponseCaching*-Middleware, die *Vary*-Eigenschaften dagegen nicht. Diese werden in der Middleware verarbeitet.

Zur Auswahl stehen die Eigenschaften *VaryByQueryKeys* und *VaryByHeader*. Die erste Variante hört auf festgelegte Abfrage-String-Keys, die zweite auf einen bestimmten Header, den der Client mitsenden muss. Um dies zu testen, kann ein neues *CacheProfile* angelegt werden:

Listing 3: MemoryCache

```
public IActionResult Company()
{
  const string cacheKey = "MyCounter";
  long cachedCounter;
  if (!_cache.TryGetValue(cacheKey,
    out cachedCounter))
  {
    var counter = _pageCounter.Count("Company");
    var options = new MemoryCacheEntryOptions
    {
      SlidingExpiration = TimeSpan.FromSeconds(30)
    };
    _cache.Set(cacheKey, counter, options);
  }

  ViewData["Message"] = "Your company page.";
  ViewData["Counter"] = cachedCounter;
  ViewData["Time"] =
    DateTime.Now.ToLongTimeString();

  return View();
}
```

```
options.CacheProfiles.Add("vary-by-version",
  new CacheProfile()
{
  Duration = 120,
  VaryByQueryKeys = new[] { "firstname", "lastname" }
});
```

Dieses Profil hält den Cache für 120 Sekunden oder bis sich der Wert des Abfrage-Strings *firstname* oder *lastname* ändert. Ruft man einen alten Wert wieder auf, wird ein älterer Cache erneut geladen. Das heißt, es wird für jeden Wert ein neuer Cache angelegt, der in diesem Fall 120 Sekunden gültig ist.

Wann ist das sinnvoll? Zum Beispiel wenn man Abfragen an einen Server schickt, die bestimmte Daten anhand der Abfrage-String-Argumente filtern. Die Anfragen lassen sich so eine gewisse Zeit zwischenspeichern.

Der Applikations-Cache

Die Eigenschaften *VaryByQueryKeys* und *VaryByHeader* sind recht interessant und ermöglichen ein spezifisches Caching. Weit öfter ist es auch sinnvoll, in der Applikation Daten für die applikationseigene Logik im Cache vorzuhalten. Dafür ist eine Cache-Schnittstelle nötig.

Für den einfachsten Fall gibt es ein Object vom Typ *IMemoryCache*, das sich per Dependency Injection in den Konstruktor eines Controllers, eines Dienstes oder eines beliebigen

anderen Objekts injizieren lässt. Zuvor muss es für den Dependency-Injection-Container in *services* registriert werden:

```
services.AddMemoryCache();
```

Für Anwendungen, die auf mehreren Docker-Containern oder Azure-App-Services-Instanzen laufen, gibt es zudem noch Caches vom Typ *IDistributedCache*, die entweder ein *MemoryDistributedCache* sein können, ein *SqlServerCache* oder ein *RedisCache*. Somit können diese Instanzen auf einen gemeinsamen Cache zurückgreifen. Diese müssen ebenfalls in der *Startup.cs*-Datei per Dependency Injection bereitgestellt und konfiguriert werden:

```
services.AddDistributedMemoryCache();
services.AddDistributedRedisCache(options => { });
services.AddDistributedSqlServerCache(options => { });
```

Diese drei verteilten Caches sollte es nicht parallel nebeneinander geben, da es nur ein *IDistributedCache*-Interface gibt.

Um das Cache-Objekt zu testen, kann eine weitere View angelegt werden, beispielsweise mit dem Namen *Company*. Dort wird der gleiche HTML-Code verwendet wie zuvor in der *About*- oder *Contact*-Sicht.

Die passende Aktion ist in Listing 3 zu sehen. Mit dem *Try*-Pattern wird über die Methode *_cache.TryGetValue<T>()* der gespeicherte Wert aus dem Speicher geladen. Wenn kein Cache angelegt wurde, wird ein neuer Wert mit der Methode *_cache.Set<T>()* erzeugt. Neben der hier verwenden Eigenschaft *SlidingExpiration* steht auch *AbsoluteExpiration* zur Verfügung, die ein *DateTimeOffset*-Objekt erwartet, sowie *AbsoluteExpirationRelativeToNow*, womit per *TimeSpan* ein fixer Zeitpunkt in der Zukunft ab jetzt festgelegt werden kann. Diese Schreibweise lässt sich mit der Methode *_cache.GetOrCreate<T>()* etwas vereinfachen, siehe Listing 4.

Mit genau diesem Cache lassen sich nun feinkörnig alle Zugriffe auf externe Datenquellen oder auf rechenintensive Operationen zwischenspeichern.

Aus bestimmten Kriterien zusammengesetzte Cache-Keys helfen dabei, für bestimmte Bedingungen eigene Caches zu erstellen. Dafür würde man zum Beispiel für Datenbankabfragen die *WHERE*-Bedingungen oder eventuelle Sortierungen in den Cache-Key mit einbauen. Auf diese Art erhält man für jede Variation einen eigenen Cache-Key und somit einen eigenen Cache-Eintrag.

Die verteilten Caches verhalten sich ähnlich und werden hier nicht separat besprochen. Der Unterschied liegt in den unterschiedlichen Konfigurationen. Außerdem bieten diese Caches die Möglichkeit, asynchron zu arbeiten; das ist sinnvoll bei Caches, die mit einer externen Datenbank arbeiten.

Cache-Tag-Helper

Zu guter Letzt gibt es noch die Möglichkeit, einzelne Teile einer View zu cachen. Mithilfe von *CacheTagHelper* oder des *DistributedCacheTagHelper* lässt sich genau das umsetzen.

Fügt man der *About*-View den folgenden Code hinzu, lassen sich gecachter und ungecachter Output vergleichen:

Listing 4: MemoryCache mit GetOrCreate()

```
long cachedCounter = _cache.
GetOrCreate<long>(cacheKey, entry =>
{
  entry.SetOptions(new MemoryCacheEntryOptions
  {
    SlidingExpiration = TimeSpan.FromSeconds(30),
    AbsoluteExpiration = DateTimeOffset.Now,
    AbsoluteExpirationRelativeToNow =
      TimeSpan.FromSeconds(39),
  });
  var counter = _pageCounter.Count("Company");
  return counter;
});
```

```
<cache expires-after="TimeSpan.FromSeconds(30)"
    enabled="true">
  <p>@ViewData["Counter"]</p>
  <p>@ViewData["Time"]</p>
</cache>
```

Beide Tag-Helper bieten die gleichen Verfallseinstellungen wie in den vorherigen Beispielen, allerdings ein paar mehr *Vary*-Eigenschaften als der Response-Cache per C#. Zusätzlich zu *QueryString* und *Header* stehen hier *vary-by-cookie*, *vary-by-user* und *vary-by-route* zur Verfügung. Diese sehr praktischen Eigenschaften werden hoffentlich auch bald in den C#-Konfigurationen verfügbar sein.

Zusammenfassung

Wie man sieht, gibt es verschiedenste Arten von Caches, die in ASP.NET Core verwendet werden können. Auch hier zeigt sich wieder der Vorteil von Dependency Injection, das den Zugriff auf Caches überall in der Applikation eröffnet. In Zeiten von Docker-Containern und Cloud-Hosting hat Microsoft natürlich auch daran gedacht, die Unterstützung von verteilten Caches zu vereinfachen und direkt zu unterstützen. ■

[1] In-memory caching in ASP.NET Core, www.dotnetpro.de/SL1802ASPCoreCache1
[2] Zwischenspeichern von Antworten in ASP.NET Core, www.dotnetpro.de/SL1802ASPCoreCache2

Jürgen Gutsch
ist Software Developer, Berater, Trainer und freier Autor. Er betreibt einen Blog auf http://asp.net-hacker.rocks, engagiert sich in der .NET-Community und wurde mehrfach als MVP ausgezeichnet. Sie erreichen ihn unter **juergen@gutsch-online.de**.

dnpCode A1802ASPCoreCache