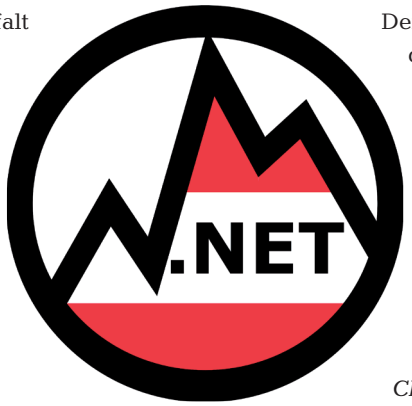
MICROSOFT.EXTENSIONS.PRIMITIVES

Die vergessene Bibliothek

Sie wird oft übersehen, und doch stellt sie die Basis vieler Frameworks.

Microsoft bietet inzwischen eine Vielfalt an Bibliotheken, die kaum überschaubar ist. Auch in dieser Kolumne waren die Bibliotheken schon häufiger Thema. Zum Beispiel haben wir uns die beliebten Logging- oder auch Dependency-Injection-Pakete näher angesehen [1]. Doch eine Bibliothek haben wir noch nicht beleuchtet: Microsoft.Extensions.Primitives.



Primitives? Was ist das?

Hier hat Microsoft eine Hilfsbibliothek entwickelt, die sie nicht Microsoft.Extensions.Utilities nennen wollten, obwohl der Name passend gewesen wäre – und deswegen den Namen Primitives gewählt. Eine einfache Installation mittels NuGet legt alle Funktionen offen:

```
<ItemGroup>
  <PackageReference
    Include="Microsoft.Extensions.Primitives"
    Version="7.0.0" />
</ItemGroup>
```

Nachdem wir die Installation durchgeführt haben, lässt sich der Namespace durchwühlen: Was hat Microsoft hier nun geliefert? Tatsächlich offenbaren sich eine Hand voll Klassen und Datentypen – nämlich vorrangig String-Zerlegung und Änderungsbenachrichtigung (Bild 1). Gerade String-Values sind normalerweise Datentypen, die man im ASP.NET-Core-Bereich vorfindet. Auch hier handelt es sich offenbar um eine Grundbibliothek für andere Frameworks von Microsoft.

Änderungsbenachrichtigung – Change Notificiations

Die Basis hat Microsoft im Interface *IChangeToken* definiert. Dieses Interface beschreibt, dass die Implementation angeben muss, ob sich etwas geändert hat beziehungsweise ob sich hierfür aktuell jemand interessiert.

Es gibt mittels der Methode *RegisterChangeCallback* die Möglichkeit, eine Methode anzugeben, welche bei einer Änderung aufgerufen wird. Das Spannende dabei ist, dass ein *IDisposable* zurückgegeben wird und somit ermöglicht wird, nach erfolgtem *using* (oder Dispose-Aufruf) eine automatische Deregistrierung vorzunehmen.

Somit lösen sich alle Callbacks bei sorgfältigem Programmieren automatisch in Luft auf. Das ist im Vergleich zu Events ein immenser Vorteil, da man hier ebenso säuberlich auf die Deregistrierung achten müsste wie darauf, dass das ein oder andere Objekt nicht im Arbeitsspeicher hängen bleibt.

Die Standardimplementierung mit CancellationChangeToken und CompositeChangeToken

Microsoft liefert uns außerdem zwei Implementierungen: *CancellationChangeToken*, das ein *CancellationToken* auf Änderungen überwacht, und *CompositeChangeToken*, welches verschiedene *IChangeToken* überwachen kann und somit mehrere Änderungen aggregiert.

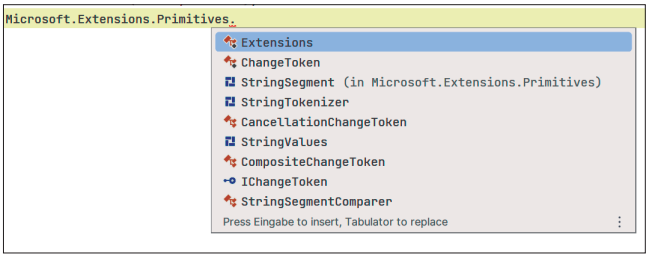
Die Klasse *CancellationChangeToken* benötigt dabei ein *CancellationToken*. Die Aufgabe dieser Hilfsklasse ist es, einen Abbruch zu überwachen, damit wir einen Callback registrieren können.

```
var cts = new CancellationTokenSource();
var cancellationChangeToken =
  new CancellationChangeToken(cts.Token);

using (cancellationChangeToken.RegisterChangeCallback(
    _ => Console.WriteLine("CTS abgebrochen.", null))
{
  await.Task.Delay(1000);
  cts.Cancel();
}
```

Dies zeigt uns der Beispielcode schon recht gut. Dieser meldet ein Token von einer *CancellationTokenSource* an und registriert einen *ChangeCallback*.

Dieser Callback wird nach einer Sekunde automatisch aufgerufen, weil ein simulierter Abbruch einer *CancellationTokenSource* ausgeführt wird. Ebenfalls möglich ist es, dass wir mehrere *IChangeToken*-Instanzen gesammelt überwachen.



IntelliSense weiß, was sich in den Primitives verbirgt (Bild 1)

```
var cts1 = new CancellationTokenSource();
var cts2 = new CancellationTokenSource();

var ctsToken1 =
  new CancellationChangeToken(cts1.Token);
var ctsToken2 =
  new CancellationChangeToken(cts2.Token):

var combined = new CompositeChangeToken(new[]
{
  ctsToken1,
  ctsToken2
});
using (combined.RegisterChangeCallback(
    _ => Console.WriteLine("CTS abgebrochen.",
    null))
{
  await.Task.Delay(1000);
  cts1.Cancel();
}
```

Dieser Code ist in der Lage, mehrere *IChangeToken* entgegenzunehmen. Exemplarisch stehen hierfür zwei *CancellationChangeToken*. Hintergrund ist dabei nun, dass sich jemand benachrichtigen lassen kann, sobald irgendein Token eine Änderung meldet.

Wer nun der Meinung ist, dass ihm dieser Fall noch nicht untergekommen ist, der sei gewarnt: Diese Klasse ist die Basis für viele andere Frameworks. Deshalb tauchen *IChangeToken* immer wieder auf.

Gerade wenn es zum Beispiel um die Änderungsbenachrichtigung der *IConfiguration* geht, wird intern immer auf diese Basis zurückgegriffen. Daher ist es sinnvoll, auch über diese Infrastrukturkomponente Bescheid zu wissen.

StringTokenizer

Eine Klasse, deren Kenntnis dem Autor dieses Artikels schon früher geholfen hätte, wenn er von ihr gewusst hätte, ist die unscheinbare Klasse *StringTokenizer*. Diese Klasse ist in der Lage, einen Text nach Trennzeichen aufzutrennen. Der Prozess läuft dabei speicheroptimiert ab.

Wie oft haben Sie Code geschrieben, mit dem Sie mittels *Strings.Split* einen Text zerlegt haben? Dass dies nicht gerade speicherschonend ist, war immer klar. Nun hat Microsoft hierfür eine wunderschöne Hilfsklasse angelegt. Mit dem *StringTokenizer* kann man eine solchen Typ instanzieren und anschließend über die Teile iterieren (Bild 2).

```
Hallo
Welt
-
Österreich
ist
ein
schönes
Land,
aber
Deutschland
ist
auch
nicht
schlecht.
```

Das Ergebnis: Viele einzelne Wörter (Bild 2)

```
var text = "Hallo Welt - Österreich ist ein " +
  "schönes Land, aber Deutschland ist auch " +
  " nicht schlecht";
StringTokenizer tokenizer =
  new StringTokenizer(text, new[] {, ,});
foreach (var vard in tokenizer)
{
  Console.WriteLine(word);
}
```

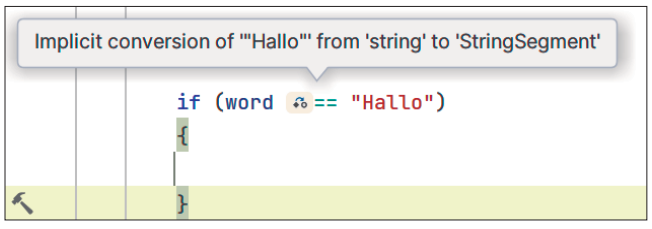
Spannend ist dabei, dass man mehrere Trenner angeben kann: Leerzeichen, Zeilenumbrüche, Tabstops oder Semikolons – je nachdem, was die Situation gerade erfordert. Das Interessante dabei ist aber, dass wir als Ergebnis nicht mehrere Strings erhalten, sondern speicherschonend ein *StringSegment*.

Ein *StringSegment* ist eine Hilfsklasse ähnlich wie ein *Span<T>*. Wir bekommen nicht sofort einen neuen String, sondern eine Art Fenster auf den Text.

Es enthält die Information, wo im originalen String sich der Text wirklich befindet. Mit anderen Worten, es wird mit unserem Code kein einziger neuer String allokiert (mit Ausnahme vom Originaltext). Somit können wir sehr effizient mit den Daten arbeiten.

Selbst ein Inhaltsvergleich macht sichtbar, dass hier nicht mit dem Datentyp *string* gearbeitet wird, sondern mit einem *StringSegment* (Bild 3).

Microsoft hat dazu die Geschwindigkeit der String-Operationen verglichen und festgestellt, dass ausgerechnet die beliebte *String.Split*-Variante massiv langsamer ist im Vergleich zum *StringTokenizer* [2].



Der Inhaltsvergleich läuft mithilfe des Typs StringSegment (Bild 3)

Fazit

Zugegeben: Der totale Reißer ist die *Microsoft.Extensions.Primitives* nicht. Dennoch ist die Bibliothek eine oft übersehene und stellt immerhin die Basis von vielen Frameworks aus dem Hause Microsoft dar. Und wir wissen alle: Je besser die Basis, umso besser ist unsere Software. ■

[1] Christian Giesswein, One Host to rule them all, dotnetpro 10/2022, Seite 46 ff., www.dotnetpro.de/A2210NETirol

[2] Benchmark comparing StringTokenizer to string.Split, www.dotnetpro.de/SL2308NETirol1

Christian Giesswein
studierte Wirtschaftsinformatik in Wien und entwickelt von klein auf Software mit .NET und C#. In Tirol hat er das Unternehmen Giesswein Software-Solutions gegründet, das sich auf Individualsoftware und Consulting spezialisiert.
christian@software.tirol

dnpCode A2308NETirol