WEB-APPS MIT BLAZOR, TEIL 1

Mit .NET ins Web

Blazor kommt nach und nach in der Praxis an, um mit .NET und C# Single-Page-Applikationen zu erstellen. Ist es eine gute Alternative zu JavaScript und Co.?

Noch immer gibt es eine gewisse Spaltung im Lager der Softwareentwickler. Die einen bauen Desktop-Applikationen und schwören auf native Apps, die auf den Endgeräten direkt und ohne Umweg über den Browser ausgeführt werden. Andere Entwickler erstellen Webapplikationen, heute oft in Form einer Single-Page-Applikation (SPA). So unterschiedlich, wie diese beiden Typen von Anwendungen sind, so heterogen sind auch die Abläufe, Programmiersprachen und Werkzeuge für die Programmierung. Die Entwicklung von Desktop-Applikationen erfolgt in der Regel in umfassenden integrierten Entwicklungsumgebungen, typische Programmiersprachen sind Java, C# et cetera. Das User Interface wird oft deklarativ und oder mit Unterstützung eines grafischen Designers erstellt. Ganz anders Webapplikationen: Technologisch haben wir eigentlich nur HTML (Struktur), CSS (Layout, Design) und JavaScript (Logik, Interaktion). Schwergewichtiger ist hier vielmehr die Entscheidung, welche Teile einer Applikation auf dem Client und welche auf dem Server ausgeführt werden. Wir können hier grob zwischen den traditionellen Webapplikationen und den Formen einer SPA unterscheiden [1] (Bild 1). Das entscheidende Merkmal besteht also darin, dass bei der ersten Variante bei der Aktualisierung der View, zum Beispiel bei der Neuanzeige von Daten, diese komplett ausgetauscht wird. Eine SPA geht hier mit Fingerspitzengefühl vor, es werden also nur diejenigen Teile der Seite (Elemente, Widgets) aktualisiert, die von der Änderung betroffen sind (Bild 2). Die SPA kann daher mit einigen Vorteilen punkten:

- schnelle Ladezeiten,
- Verbesserungen bei der User Experience, denn das Handling der Web-App fühlt sich zügig an,
- vom Backend entkoppelte Frontend-Entwicklung,
- Ausnutzung des lokalen Caches für ein performantes Verhalten der Applikation.

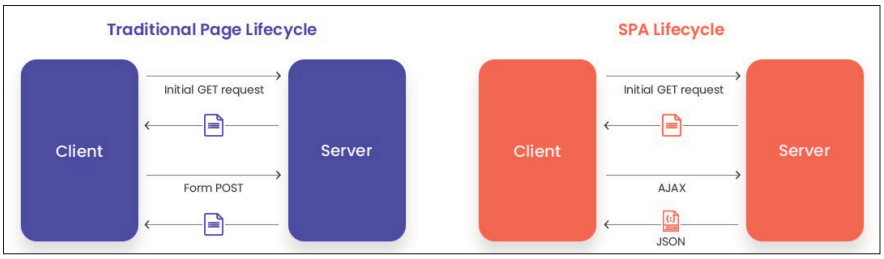


Bild: ExcellentWebworld [1]

Lifecycle einer traditionellen Webapplikation und einer SPA (Bild 1)

Diese Vorteile einer SPA sind jedoch auch nicht umsonst zu haben. Es braucht dafür eine gut durchdachte Anwendungsarchitektur.

Genau an dieser Stelle kommen Bibliotheken und Frameworks zum Einsatz, um für die Webapplikation eine Struktur vorzugeben beziehungsweise die Arbeit mit stetig wiederkehrenden Aufgaben zu erleichtern und zu standardisieren. Dazu gehören zum Beispiel Frameworks wie Angular.js oder React.js. Hier gibt es eine reiche Auswahl an Bibliotheken und Frameworks, die man darüber hinaus gefühlt alle miteinander kombinieren kann.

Auch das Tooling zum Erstellen einer solchen SPA unterscheidet sich in vielen Punkten vom Vorgehen bei der Programmierung von nativen Applikationen. Statt in einer alles umfassenden integrierten Entwicklungsumgebung mit einem „schützenden" Designer wird in einem Editor mit Plug-ins und Extensions eigener Wahl und der Interaktion auf der Kommandozeile forsch drauflosprogrammiert. Die einen lieben diese Vielfalt und Flexibilität, andere Entwickler – auch Umsteiger – sehnen sich vielleicht auch danach, etwas mehr Struktur in den Entwicklungszyklus zu bekommen. Insbesondere das Dreiergespann aus HTML, CSS und JavaScript ist nicht jedermanns Sache, denn es fokussiert sehr stark auf die technischen Belange und lenkt schnell vom Fokus der App ab.

Relativ neue Ansätze sind Blazor Server und Blazor WebAssembly [2][3]. Beide Technologien basieren auf Microsoft .NET und man programmiert mit C#. Damit dürfte die Zielgruppe von Blazor klar sein: Entwickler mit Erfahrungen in .NET und C# sollen sich hier schneller als in der klassischen Webwelt zu Hause fühlen und in der Lage sein, moderne

Web-Apps mit Blazor

Teil 1: Einordnung, Architektur und Technologie
Teil 2: Grundlagen der App-Entwicklung mit Blazor
Teil 3: Praxisbeispiel I
Teil 4: Praxisbeispiel II
Teil 5: Praxisbeispiel III

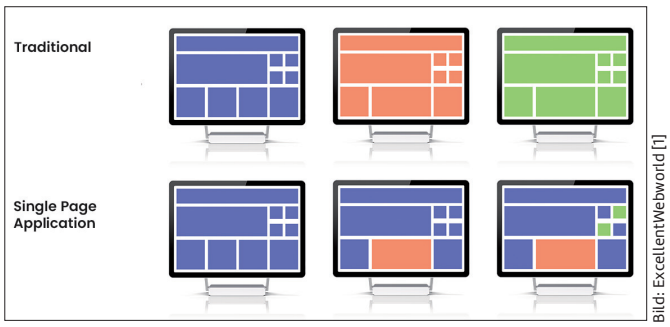


Bild: ExcellentWebworld [1]

Auswirkungen auf die View bei einem Refresh bei traditioneller Webapplikation und einer SPA (Bild 2)

SPAs zu erstellen. Das ist unser Thema für die mit diesem Artikel beginnende Serie: eine systematische Einführung mit Hintergründen und einen ersten Einblick in die Programmierpraxis zu geben, siehe Kasten Web-Apps mit Blazor. Am Ende sollte der Leser in der Lage sein, erste moderne Webapplikationen mit Blazor zu erstellen. Gelingt dies, dann erweitern wir die Reichweite der von uns eingesetzten Technologie erheblich. Wir können dann mit .NET und C# nicht nur nativ auf dem Desktop (UWP, Win UI, WPF, Win Forms) und mobil (Xamarin) punkten, sondern erreichen auch das Web.

Technischer Background

Bei Blazor handelt es sich um ein Open-Source-Framework von Microsoft. Es verwendet wie andere Webframeworks HTML und CSS zur Darstellung. Ein Unterschied besteht jedoch darin, dass die Logik nicht mit JavaScript, sondern mit C# und der Razor-Syntax ausgeführt wird. Bevor wir also konkret einsteigen können, müssen wir diese technischen Hintergründe aufklären. Die Basis für den neuen Ansatz ist die WebAssembly-Technologie (Wasm) [4]. WebAssembly ist ein binäres Befehlsformat für die virtuelle Maschine im Browser. Es wurde als portables Kompilierungsziel für Programmiersprachen entwickelt und ermöglicht die Bereitstellung im Web für Client- und Serveranwendungen. Ziel der Entwicklung ist eine Ergänzung von performanten Funktionen zu JavaScript, sowohl was die Ladezeiten als auch die Ausführung betrifft. Das Projekt wird von allen großen Entwicklern der Browser-Engines, also Mozilla, Microsoft, Google und Apple, betrieben. Was sind die Gründe für diesen neuen Standard? WebAssembly bietet zwei Hauptvorteile:

- Das Binärformat, das für WebAssembly angewendet wird, kann viel schneller decodiert werden als JavaScript-Code. Experimente zeigen, dass der Code bis zu 20-mal schneller ausgeführt wird. Auf Mobilgeräten kann dieser Vorteil noch größer sein. Der Schub an Performance verbessert die Benutzererfahrung.
- Der neue Standard erleichtert das Hinzufügen von neuen Funktionen.

Natürlich bringt jeder neue Standard auch die Notwendigkeit von Anpassungen mit sich, dennoch geht man davon aus, dass langfristig die Vorteile die Nachteile übersteigen. Die Unterschiede in der Technologie sind für eine klassische JavaScript- und eine auf WebAssembly basierende Webapplikation in Bild 3 gegenübergestellt [5]. Vor der Nutzung von WebAssembly wurde der JavaScript-Code durch den Client (Browser) vom Server geladen, analysiert (Parser, Compiler) und dann ausgeführt. Mit der WebAssembly-Technologie kompiliert der Server den Code in Wasm (WebAssembly). Dieser wird dann direkt vom Just-in-Time (JIT)-Compiler des Browsers ausgeführt. Damit ist WebAssembly ein binäres Format, das für die Ausführung im Browser optimiert ist, und kein JavaScript-Code.

WebAssembly bietet zusammengefasst die folgenden Merkmale (Vorteile):

- Standardisierung: Die Entwicklung und Standardisierung wird vom W3C gesteuert.
- Breite Unterstützung: WebAssembly kann bereits heute in allen gängigen Browsern ausgeführt werden. Es ist direkter Bestandteil des Browsers, ein zusätzliches Plug-in ist nicht notwendig.
- Nutzung außerhalb des Browsers: WebAssembly wurde nicht für den Einsatz in einem Browser konzipiert, sondern kann auch für Desktop- und Mobile-Plattformen genutzt werden.
- Interaktion mit JavaScript: WebAssembly läuft in einer Art „Sandkasten“ im Browser. Eine Interaktion mit JavaScript, also ein gegenseitiger Aufruf, ist möglich. ▶

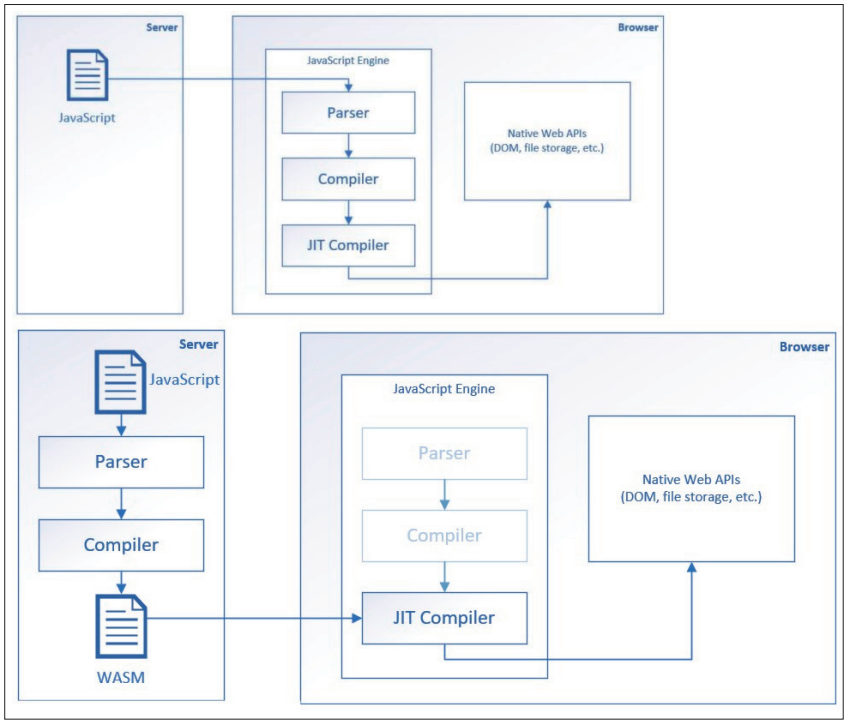


Bild: Blog.Noser.com [5]

Webapplikationen mit JavaScript versus WebAssembly (Bild 3)

- Performant: WebAssembly ist ebenso schnell wie nativer Code.
- Sprachneutral: WebAssembly ist grundsätzlich sprachneutral einsetzbar, das heißt, man kann den Quellcode einer Programmiersprache in WebAssembly kompilieren.

WebAssembly ist eine technische Basis für Blazor. Die andere Basis ist Razor. Es gilt: Browser + Razor = Blazor. Um hier also weiterzukommen, müssen wir den Ansatz von Razor zumindest fundamental erklären und verstehen.

Basis der Razor-Syntax

Razor ist eine Markup-Syntax zum Einbetten von Code (hier C#) in HTML. Razor verwendet das @-Symbol für den Übergang von HTML zu C#. Dabei werden die C#-Ausdrücke ausgewertet und zur HTML-Ausgabe verarbeitet. Man unterscheidet implizite und explizite Razor-Ausdrücke. Ein impliziter Ausdruck beginnt mit @ und es folgt der C#-Code, also zum Beispiel:

```
<p>@DateTime.Now</p>
```

In diesem Fall erfolgt also die Ausgabe des aktuellen Datums innerhalb des Absatzes (*<p/>*). Implizite Ausdrücke dürfen keine Leerzeichen und keine Generics enthalten. Hier sind dann explizite Ausdrücke zu verwenden.

Explizite Ausdrücke beginnen ebenfalls mit einem @-Zeichen, danach folgt eine öffnende Klammer, dann folgt der C#-Quellcode und wiederum eine schließende Klammer, also in der Form *@(…)*. Ein Beispiel:

```
<p>Last week this time: @(DateTime.Now -
  TimeSpan.FromDays(7))</p>
```

Jeglicher Code innerhalb von *@(…)* wird ausgewertet und in die Ausgabe gerendert.

SignalR

ASP.NET SignalR ist eine Bibliothek für ASP.NET-Entwickler, mit der man in Anwendungen Echtzeit-Webfunktionen hinzufügen kann. Das bedeutet die Fähigkeit, serverseitigen Code dazu zu bringen, Inhalte in Echtzeit auf die verbundenen Clients zu übertragen.

SignalR nutzt mehrere Transportkanäle und wählt automatisch die beste verfügbare Option aus. SignalR nutzt WebSocket und ein HTML5-API, das eine bidirektionale Kommunikation zwischen Browser und Server ermöglicht. SignalR bietet außerdem ein einfaches API auf hoher Ebene für die Ausführung auf der Basis des Server-zu-Client Remote Procedure Call (RPC), also zum Aufruf von JavaScript-Funktionen im Browser eines Clients über serverseitigen .NET-Code in einer ASP.NET-Anwendung sowie zum Hinzufügen nützlicher Hooks für das Verbindungsmanagement.

Listing 1: Objektdefinition und Schleife in Razor

```
@{
  var people = new Person[]
  {
    new Person("Klaus", 50),
    new Person("Johanna", 33),
  };
}

@foreach (var person in people)
{
  <p>Name: @person.Name</p>
  <p>Age: @person.Age</p>
}
```

Neben Ausdrücken können wir auch Codeblöcke definieren. Razor-Codeblöcke beginnen mit @ und werden von *{…}* eingeschlossen. Hier erfolgt kein Rendering, sondern es wird Logik definiert. Ein Beispiel:

```
@{
  void PrintName(string name)
  {
    <p>Name: <strong>@name</strong></p>
  }

  PrintName("Klaus Müller");
}
```

Wir definieren eine lokale Funktion in C# namens *PrintName(…)* mit dem Parameter *name* und dem Datentyp *string*. Diese Funktion gibt die Zeichenkette *name* innerhalb eines *<p\>*-Tags aus. Die Funktion können wir mit einem Wert des Parameters aufrufen. Im Ergebnis bekommen wir folgenden HTML-Code:

```
<p>Name: <strong>Klaus Müller</strong></p>
```

Flexibel wird das „Mischen“ von HTML und C# durch die Möglichkeit, Auswahlentscheidungen, Schleifen, Fehlerbehandlung und mehr nutzen zu können. Dazu noch ein kleines Beispiel (Listing 1): Wir definieren zunächst eine Liste von Objekten mit dem Datentyp *Person*, das heißt, die Klasse *Person* sollte schon existieren. Im nächsten Schritt iterieren wir über jedes Objekt der Liste mithilfe der *Foreach*-Schleife. Der Quellcode ist selbsterklärend und entspricht dem Vorgehen in C#. Die Razor-Syntax holt also die Sprachmerkmale von C# in den HTML-Code und macht diesen damit dynamisch.

Die weiteren Sprachmerkmale funktionieren ähnlich. Einen kompakten Überblick über die Razor-Syntax bekommen Sie beispielsweise in der Dokumentation unter [6].

Wenn Sie (noch) keine Erfahrungen mit Razor gesammelt haben, so ist das an dieser Stelle kein Problem. Bei der Ent-

wicklung mit Blazor benötigen wir zwar diese Syntax, aber wir können sie durchaus dann implizit mit erlernen, wenn wir erste Blazor-Komponenten erstellen. Weiter geht es daher mit Blazor.

Blazor-App-Typen

Blazor gibt es in mehreren Ausführungsvarianten [2] [7]:

- Blazor Server (Web-Apps)
- Blazor WebAssembly (Web-Apps)
- Blazor Electron (Desktop-Apps)
- Blazor Mobile Bindings (Mobile-Apps).

Sehen wir uns diese einzelnen Spielarten für Blazor-basierte Apps genauer an.

Die Funktionsweise von Blazor Server ist in Bild 4 zu sehen [7]. Die Arbeitsweise ist der von ASP.NET MVC und ASP.NET Razor Pages ähnlich, jedoch wird auf dem Server eine SPA gerändert und diese wird an den Client ausgeliefert. Zum Client werden JavaScript und Markup gesendet und die Daten und Benutzereingaben werden laufend mittels der Bibliothek SignalR zwischen Client und Server ausgetauscht, vergleiche den Kasten SignalR.

Das Blazor-Server-Hostingmodell besitzt folgende Vorteile:

- Geringe Downloadgröße: Die Ladezeit der App verkürzt sich gegenüber einer WebAssembly-App.
- .NET-Unterstützung: Volle Serverunterstützung inklusive aller .NET-Core-kompatiblen APIs.
- NET-Tool-Unterstützung: Diese funktioniert auf den Servern, wie zum Beispiel das Debugging.
- Breite Unterstützung: Die App läuft auch auf Servern, die WebAssembly nicht unterstützen.
- Sichere Codebasis: Die .NET-Codebasis verbleibt auf dem Server.

Wie immer gibt es auch ein paar Einschränkungen:

- eingeschränkte Teilnehmerzahlen,
- große Netzwerklatenz führt zu spürbaren Verzögerungen bei der Darstellung des User Interfaces,
- eine permanente Kommunikation zum Server vom Client ist notwendig.

Tabelle 1: Blazor Server versus Blazor WebAssembly

Merkmal	Blazor Server	Blazor WebAssembly
Volle .NET-Funktionalität	+	
Performance zur Laufzeit		+
Schneller Download bei Start der App	+	
Codesicherheit	+	
Einsatz auf leistungsschwachen Clients	+	
Offline-Betrieb		+
Skalierbarkeit		+
Zugriff auf Server-Ressourcen	+	

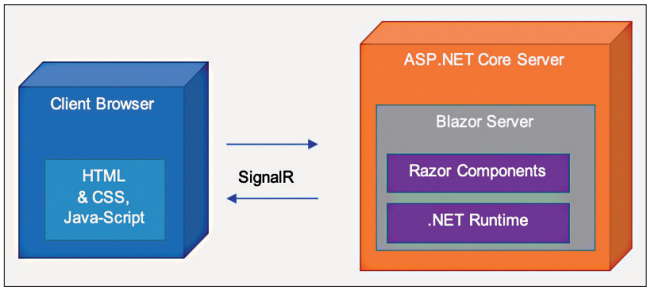


Bild: BBV.ch [7]

Architektur von Blazor Server (Bild 4)

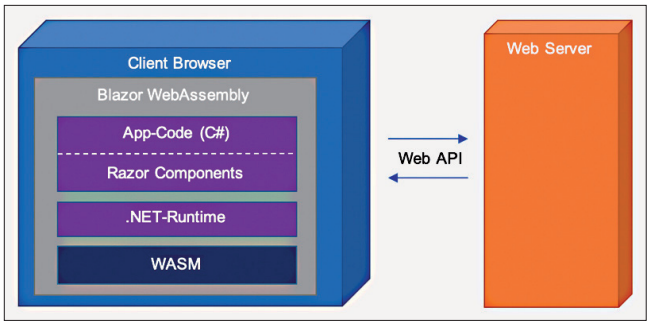


Bild: BBV.ch [7]

Architektur von Blazor WebAssembly (Bild 5)

Kommen wir zu Blazor WebAssembly. Wir sehen die Architektur in Bild 5 [7]. Blazor WebAssembly läuft auf dem Client (siehe obige Ausführungen zum Thema WebAssembly). Die Blazor-App, die jeweiligen Abhängigkeiten und die .NET-Runtime werden vom Browser heruntergeladen. Diese App wird direkt im Thread des User Interface des Browsers ausgeführt. Der Zugriff auf Kamera, Mikrofon und andere Komponenten des Clients ist ohne JavaScript über Blazor Component möglich. Gibt es diese nicht, dann kann über eine JavaScript-Interoperation mittels JavaScript zugegriffen werden. Der Webserver muss lediglich das Static File Deployment unterstützen, um eine Blazor WebAssembly zu hosten. Da der gesamte Code direkt im Client läuft, kann die Applikation gut skalieren. Lediglich der initiale Download ist groß, was zu einer längeren Startzeit führt. Danach läuft die App performant. Tabelle 1 vergleicht die beiden Ansätze von Blazor Server und Blazor WebAssembly anhand wesentlicher Merkmale miteinander.

Kommen wir zu den beiden weiteren Blazor-App-Typen. Mit Blazor Mobile Bindings soll das App-Modell speziell für Mobile Devices zugänglich gemacht werden [8]. Das Ziel besteht darin, dass Erstellen von nativen und hybriden Apps mit C# und .NET für Android und iOS unter Verwendung bekannter Web-Programmiermuster zu ermöglichen. Blazor Mobile Bindings verwendet die Razor-Syntax, um User-Interface-Komponenten und das Verhalten der App zu definieren. Die zugrunde liegenden Komponenten basieren auf nativen UI-Komponenten von Xamarin.Forms und werden in Hybrid-Apps mit HTML-Elementen gemischt. Das Ganze hat noch experimentellen Status.

Ebenfalls noch in der Phase des Probierens ist der Ansatz, eine App auf der Basis von Blazor mithilfe von Electron auf ▶

dem Desktop lauffähig zu machen. Informationen dazu finden Sie zum Beispiel unter [9] oder [10].

Wie geht es jetzt hier weiter? Wir verschaffen uns zunächst einen allgemeinen Überblick über Blazor und den zugrunde liegenden komponentenbasierten Ansatz. Danach geht es um die Installation, die Einrichtung und das Tooling, um mit dem Blazor-Framework eine App zu erstellen. Unseren ersten Teil der Serie schließen wir mit den klassischen „Hello World"-Apps ab, um einen ersten Eindruck vom Entwicklungszyklus mit Blazor zu bekommen.

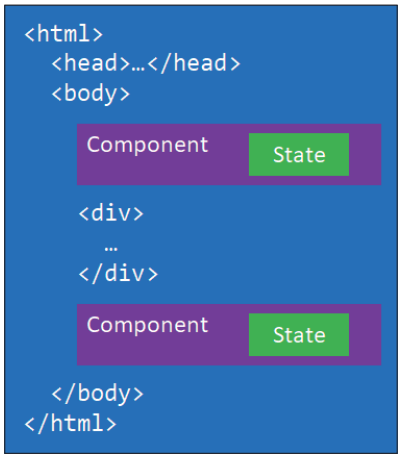


Bild: Microsoft Docs [12]

Die Blazor-Stammkomponente wird in HTML eingebunden (Bild 6)

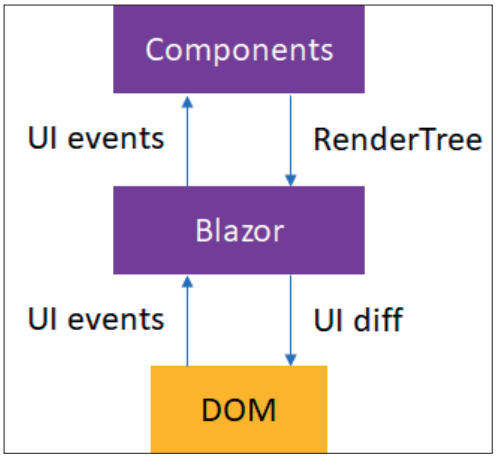
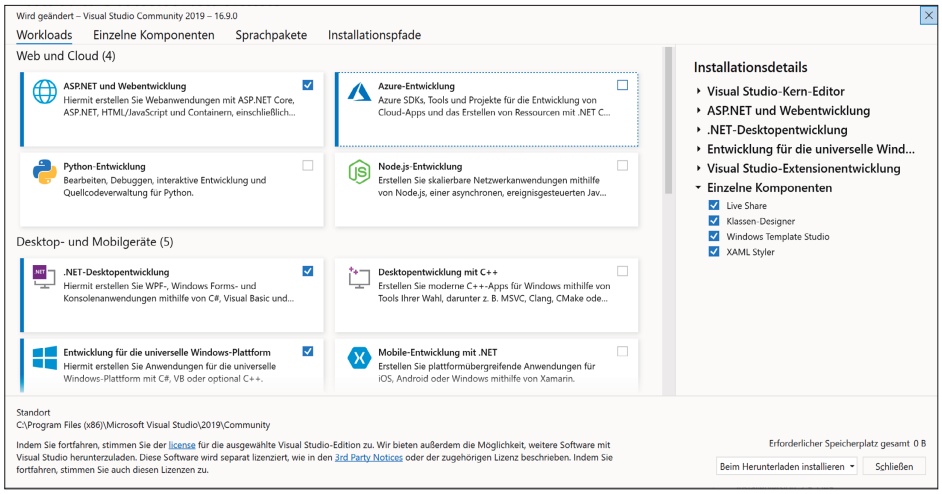


Bild: Microsoft Docs [12]

Synchronisation zwischen DOM und internem Blazor-Modell (Bild 7)



Installation des Workloads ASP.NET und Webentwicklung in Visual Studio (Bild 8)

Blazor: Überblick und Komponenten

Sehen wir uns das Framework Blazor genauer an. Das Ziel sind also moderne clientseitige Webapplikationen. Dabei haben wir es mit den folgenden Kerneigenschaften zu tun:

- Erstellen der App inklusive des User Interface mit C#.
- Client- und Serverlogik basieren auf .NET.
- Finales Rendering der Oberfläche mit HTML und CSS für eine umfassende Browserunterstützung.
- Mögliche Integration in Docker für das Hosting.
- Unterstützung für die Entwicklungsumgebungen Visual Studio, Visual Studio Mac und Visual Studio Code.

Blazor basiert auf einem komponentenbasierten Ansatz. Eine Komponente ist zum Beispiel eine Seite, ein Dialogfeld, ein Formular und so weiter. Zugrunde liegt ein Ereignismodell, um auf Benutzerinteraktionen zu reagieren.

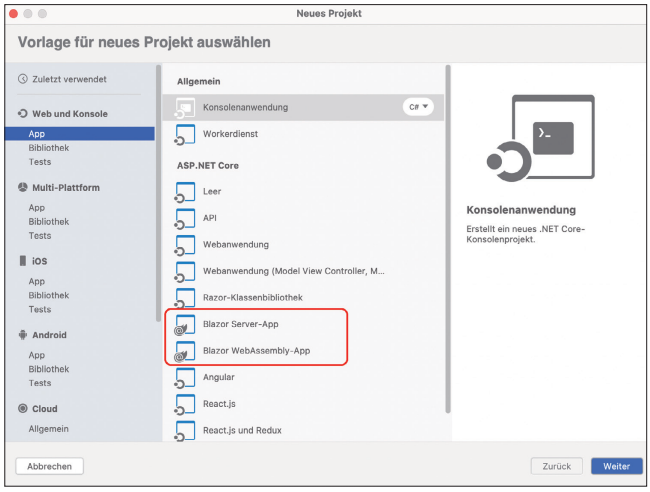
Die gesamte App besteht aus einer oder mehreren Stammkomponenten, die auf der initialen HTML-Seite eingebunden werden (Bild 6) [11]. Eine Komponente ist eine .NET-Klasse, hat einen Zustand und Logik zum Rendern. Eine Komponente kann Benutzerinteraktionen verarbeiten und dann den Status aktualisieren. Beim Rendering erfolgt ein Abgleich zwischen den DOM-Modell der Seite und einer internen Repräsentation des DOM-Modells in Blazor. Dabei werden die notwendigen Änderungen aus der Differenz der zwei Zustände bestimmt und die Änderungen umgesetzt (Bild 7) [11]. Ein manuelles Anstoßen eines Renderings einer Komponente ist möglich.

Eine Komponentenklasse hat die Dateierweiterung **.razor*. In einer solchen Datei werden HTML-Markup und C#-Quellcode miteinander kombiniert (siehe oben). Durch das Tooling der Entwicklungsumgebung wird diese Besonderheit mit IntelliSense bei der Codierung unterstützt.

Der Blick in den „gemischten" Quellcode (Listing 2) lässt uns den Aufbau einer Komponente erkennen. Es handelt sich um eine einfache Zählerkomponente. Im HTML-Teil wird das User Interface definiert. Das geschieht mit den normalen HTML-Tags (*<h1\>*, *<p\>*, *<button\>*). Im C#-Teil findet sich üblicher C#-Code. Im Beispiel werden beide Teile über ein Ereignis *(@onclick)* und die lokale Variable (*currentCount*) miteinander verbunden. Damit wird auch eine Aktualisierung des User Interface als Reaktion auf die Benutzerinteraktion erreicht.

Installation und Tooling

Installation und Tooling sind weitgehend flexibel und lassen sich mit überschaubarem Aufwand einrichten. Was die Entwicklungsumgebung angeht, haben wir je nach Betriebssystem die Wahl:



Visual Studio für Mac unterstützt beide Hosting-Modelle für Blazor (Bild 9)

- Windows: Visual Studio oder leichtgewichtiger mit Visual Studio Code.
- macOS: Visual Studio für Mac oder Visual Studio Code.
- Linux: Visual Studio Code.

Beachten Sie dabei noch die folgenden Hinweise. Beginnen wir mit Visual Studio 2019. Spielen Sie in diesem Zusammenhang die gegebenenfalls noch ausstehenden Updates ein. Ebenso installieren Sie den Workload *ASP.NET und Webentwicklung* über den Visual Studio Installer (Bild 8).

Wenn Sie unter macOS arbeiten, können Sie Visual Studio für Mac verwenden. In diesem Fall gilt: Ab Version Visual Studio für Mac 8.6 werden beide Hosting-Modelle, also sowohl Blazor WebAssembly als auch Blazor Server, unterstützt (Bild 9)

Arbeiten Sie mit einer der beiden Visual-Studio-Versionen, sind Sie gleich bereit für Ihre Blazor-App. Kommen wir zu Visual Studio Code. Aktualisieren Sie auch hier auf die neueste Version. Dieser Editor ist bekanntlich flexibel anpassbar und betriebssystemneutral. Wir müssen die aktuelle Version des .NET Core SDK installieren [12]. Prüfen Sie mit der Eingabe von

```
dotnet -version
```

auf der Kommandozeile die Aktualität des SDK. Installieren Sie die Erweiterung *C# for Visual Studio Code* [13] für eine Unterstützung der Programmiersprache C#. Ebenso ist es sinnvoll, die Erweiterung *JavaScript-Debugger* zu installieren und diese zu aktivieren (Bild 10). Damit ist die Einrichtung abgeschlossen.

Arbeitet man mit Visual Studio Code, wird das Projektskelett auf Ebene der Kommandozeile angelegt. Dazu kommen wir gleich noch einmal.

Wir wollen im Folgenden davon ausgehen, dass wir unter dem „großen" Visual Studio arbeiten. ▶

Listing 2: Aufbau einer Razor-Komponente

```
@page "/counter"
<h1>Counter</h1>
<p>Current count: @currentCount</p>
<button class="btn btn-primary" @onclick=
  "IncrementCount">Click me</button>
@code {
  private int currentCount = 0;

  void IncrementCount()
  {
    currentCount++;
  }
}
```

Blazor-App auf der Kommandozeile

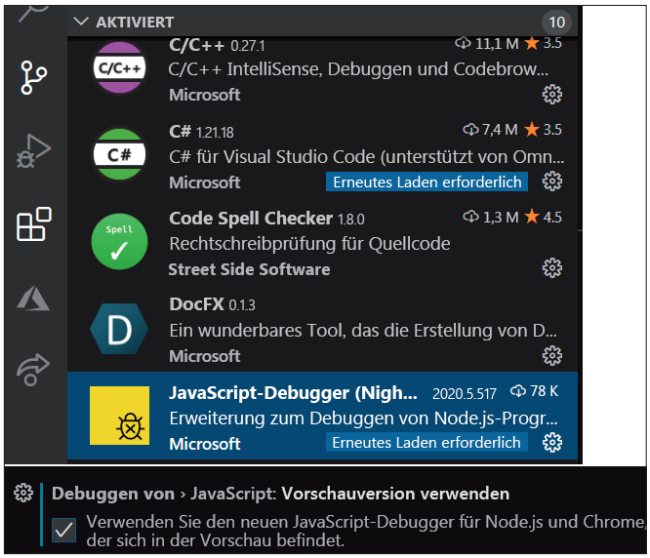
Möchten Sie nicht Visual Studio für die Programmierung verwenden, dann können Sie das Projekt auf der Kommandozeile erstellen. Für eine Blazor-WebAssembly-App führen Sie den folgenden Befehl in einer Shell aus:

```
dotnet new blazorwasm -o WebApplication1
```

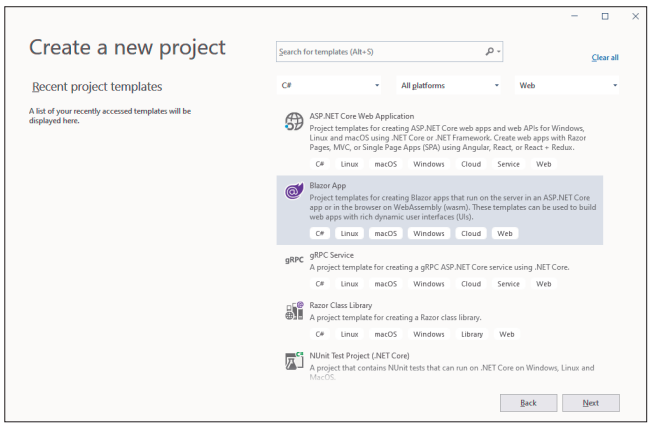
Für eine Blazor-Server-App lautet der Befehl

```
dotnet new blazorserver -o WebApplication1
```

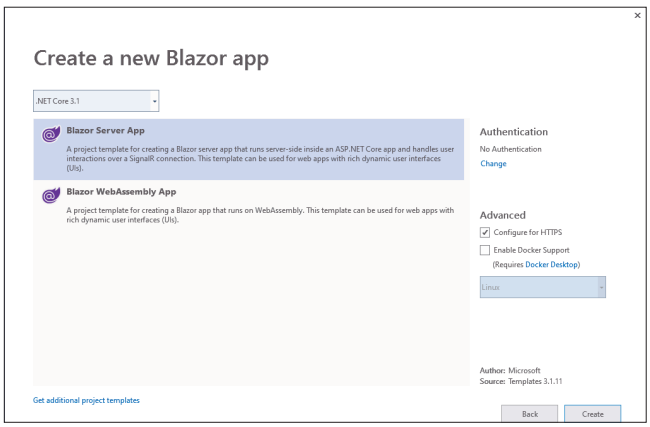
Auf diese Befehle hin wird die App generiert und kann dann im Editor, zum Beispiel Visual Studio Code, geöffnet werden.



Einrichtung von Visual Studio Code für die Entwicklung von Blazor-Apps (Bild 10)



Neue Blazor-App in Visual Studio erstellen (Bild 11)



Wahl des Hosting-Modells in Visual Studio (Bild 12)

„Hello World" für Blazor WebAssembly

Los geht es. Starten wir Visual Studio 2019 und erstellen wir ein neues Projekt des Typs *Blazor App* (Bild 11). Als Nächstes werden wir aufgefordert, das Hosting-Modell zu wählen (Bild 12). In diesem Fall entscheiden wir uns für *Blazor WebAssembly*. Danach erstellt Visual Studio die App mit einigen Beispielkomponenten und konfiguriert diese entsprechend, das heißt, dass unter anderem IIS Express als lokaler Webserver eingerichtet wird.

Sind wir einmal so weit gekommen, juckt es uns in den Fingern, auf *Ausführen* zu klicken. Das machen wir auch und starten die App direkt aus Visual Studio heraus. Darüber hi-

Tabelle 2: Projektstruktur einer Blazor-WebAssembly-App

Ordner / Datei	Blazor WebAssembly	Blazor Server
Pages	Dieser Ordner enthält die routingfähigen Komponenten / Seiten (*.razor*), aus denen die Blazor-App besteht.	
Properties / launchSettings.json	Enthält die Konfiguration der Entwicklungsumgebung.	
Shared-Ordner	Enthält die freigegebenen Komponenten und Stylesheets.	
wwwroot	Das Webstammverzeichnis für die App, das die öffentlichen statischen Ressourcen der App enthält (appsettings.json). Die Seite *index.html* ist die Stammseite der App. Wenn eine Seite der App zum ersten Mal angefordert wird, wird diese Seite gerendert und in der Antwort zurückgegeben. Die Seite gibt an, wo die App-Stammkomponente gerendert wird. Die Komponente wird an der Position des *div*-Elements mit einem *id*-Objekt von *app* gerendert.	Der Ordner *Web Root* für die App, der die öffentlichen statischen Ressourcen der App enthält.
_Imports.razor	Enthält die Razor-Anweisungen, die in die Komponenten der App (*.razor*) eingefügt werden sollen, zum Beispiel *@using*-Anweisungen für Namespaces.	
appsettings.json	–	Konfigurationseinstellungen für die App.
Startup.cs	–	Enthält die Startlogik der App. Zwei Methoden werden definiert: • *ConfigureServices*: Diese Methode konfiguriert die Dependency Injection-Dienste der App. • *Configure*: Konfiguriert die Pipeline für die Anforderungsverarbeitung der App. *MapBlazorHub* wird aufgerufen, um einen Endpunkt für die Echtzeitverbindung mit dem Browser einzurichten. Die Verbindung wird mit SignalR hergestellt.
App.razor	Stammkomponente der App, die das clientseitige Routing mithilfe der Router-Komponente einrichtet. Die Router-Komponente fängt die Browsernavigation ab und rendert die angeforderte Seite.	–
Program.cs	Der Einstiegspunkt der App, der den WebAssembly-Host einrichtet.	

Quelle: Microsoft Docs [14]

naus wird noch die eine oder andere Berechtigung angefordert und die App im Browser (Edge) gestartet. Der Server lauscht dabei auf der lokalen Adresse *https://localhost* auf dem Port *44339*.

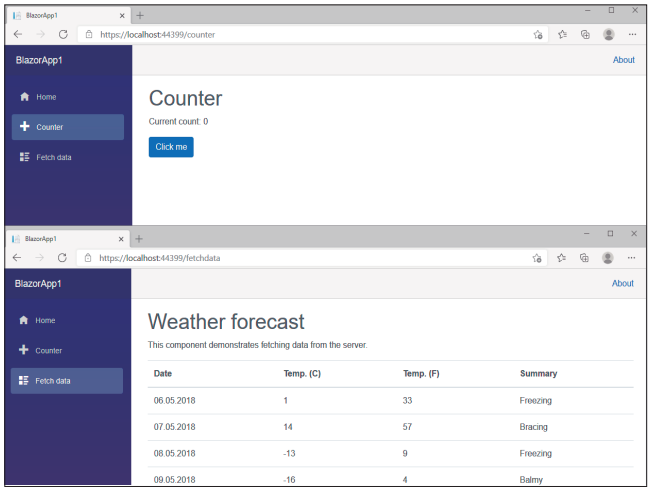
Die Beispiel-App umfasst drei Komponenten, die in Form von Pages dargestellt werden; konkret sind dies eine statische Seite (Home), eine Seite mit Interaktion (Button mit Klick-Funktion) und eine Seite mit einer tabellarischen Darstellung (Grid) (Bild 13).

Als Voraussetzung für den weiteren Einstieg studieren wir zunächst die generierte Projektstruktur der Blazor-WebAssembly-App (vergleiche Tabelle 2). Anschließend kommen wir zum Hosting-Modell Blazor Server.

„Hello World" für Blazor Server

Durchlaufen wir nun den Vorgang zum Erstellen der Blazor-App für das Hosting-Modell auf dem Server. Auch mit dieser Vorlage wird das Anwendungsgerüst erstellt und eine startfähige App generiert.

Nach dem Build und dem Start der App sehen wir gegenüber dem Anwendungsmodell Blazor WebAssembly keinen Unterschied. Dieser findet sich ausschließlich unter der Haube, nämlich in der in Tabelle 2 vorgestellten Aufteilung der Aufgaben zwischen Client (Browser) und dem Server. Einige Unterschiede gibt es in der Dateistruktur.



Eine erste Blazor-WebAssembly-App (Bild 13)

Mit diesen beiden App-Modellen haben wir den Anfang geschafft und können uns dann konkreten und typischen Aufgaben der App-Entwicklung zuwenden.

Fazit und Ausblick

In diesem ersten Teil der Artikelserie haben wir Blazor in die Welt der modernen Webprogrammierung eingeordnet. Programmieransatz und Tooling richten sich klar an Entwickler mit Kenntnissen aus dem .NET-Umfeld.

Auch die Nutzung von WebAssembly scheint ein gangbarer Weg zu sein, da hier keine herstellerspezifischen Plugins benötigt werden, sondern es auf einem Webstandard aufgesetzt. Auch wenn dieser zugegebenermaßen noch neu ist, macht es einen erheblichen Unterschied.

Im nächsten Teil der Serie zu Blazor wenden wir uns den Basics zu, die wir zum Erstellen von Blazor-basierten Apps benötigen. Das sind beispielsweise das Erstellen von Komponenten, das Routing, die Datenbindung und das Behandeln von Ereignissen. Wenn wir das alles zusammenhaben, sind wir so weit, uns an etwas größere Projekte zu wagen. ■

[1] Excellent Webworld, What is a Single Page Application?, www.dotnetpro.de/SL2105BlazorLernen1

[2] Microsoft Docs, Introduction to ASP.NET Core Blazor, www.dotnetpro.de/SL2105BlazorLernen2

[3] Microsoft .NET, Blazor, www.dotnetpro.de/SL2105BlazorLernen3

[4] WebAssembly, https://webassembly.org

[5] Noser Engineering, Einführung in Microsoft ASP.NET Blazor, www.dotnetpro.de/SL2105BlazorLernen4

[6] Microsoft Docs, Razor syntax reference for ASP.NET Core, www.dotnetpro.de/SL2105BlazorLernen5

[7] bbv Software Services AG, Hier kommt Blazor, www.bbv.ch/blazor/

[8] Microsoft Docs, Experimental Mobile Blazor Bindings, www.dotnetpro.de/SL2105BlazorLernen6

[9] JetBrains Blog, Run Blazor Apps Within Electron Shell, www.dotnetpro.de/SL2105BlazorLernen7

[10] Steve Sanderson's Blog, Exploring lighter alternatives to Electron for hosting a Blazor desktop app, www.dotnetpro.de/SL2105BlazorLernen8

[11] Microsoft Docs, Architecture comparison of ASP.NET Web Forms and Blazor, www.dotnetpro.de/SL2105BlazorLernen9

[12] .NET-Downloads, https://dotnet.microsoft.com/download

[13] C# for Visual Studio Code, www.dotnetpro.de/SL2105BlazorLernen10

[14] Microsoft Docs, ASP.NET Core Blazor project structure, www.dotnetpro.de/SL2105BlazorLernen11

Elena Bochkor
arbeitet am Entwurf und Design mobiler Anwendungen und Webseiten. Weitere Informationen zu diesen und anderen Themen der IT finden Sie unter https://larinet.com. Folgen Sie ihr auf Instagram unter **www.instagram.com/larinetcommunication**.

Dr. Veikko Krypczyk
ist begeisterter Entwickler und Fachautor. Weitere Informationen zu diesen und anderen Themen der IT finden Sie unter https://larinet.com. Folgen Sie ihm auf Instagram unter **www.instagram.com/larinetcommunication**.

dnpCode	A2105BlazorLernen