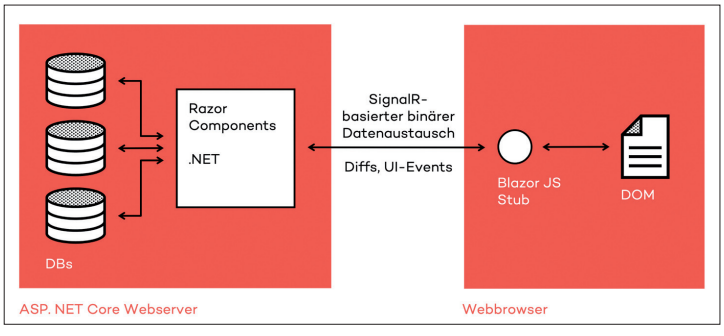
PRAKTISCHE ERFAHRUNGEN MIT BLAZOR

Zwischenbilanz

Blazor WebAssembly in der Praxis: Reife, gelungener Einsatz, Showstopper.

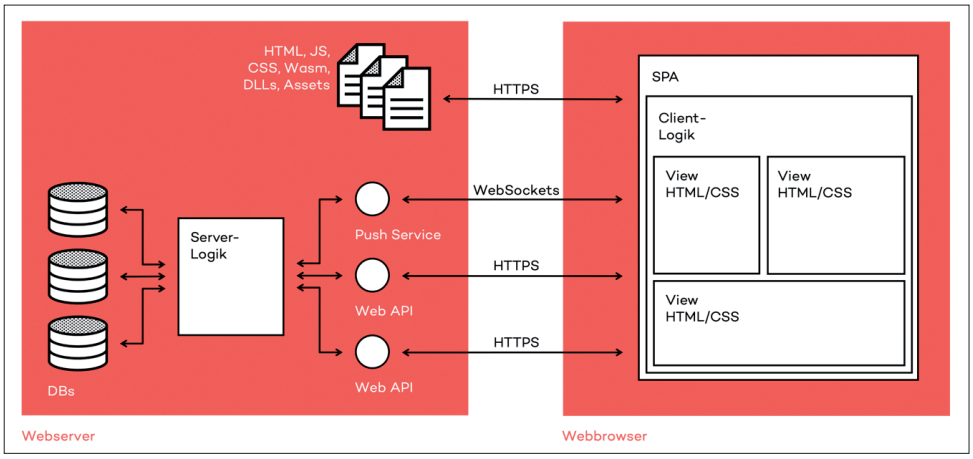
ASP.NET Core Blazor ist Microsofts Framework zur Implementierung webbasierter Anwendungen, das sich an Entwickler mit Kenntnissen in .NET und C# richtet. Es besteht neben anderen Frameworks wie ASP.NET Core MVC. Rund zweieinhalb Jahre nach der Veröffentlichung von Blazor WebAssembly und auf Basis unserer Erfahrungen aus vielen Kundenprojekten bei Thinktecture wollen wir eine Zwischenbilanz ziehen: Wie ist der aktuelle Zustand des Frameworks? Wie gelingt der Einsatz von Blazor? Und wo stößt es an seine Grenzen?



Funktionsweise einer Blazor-Server-Anwendung (Bild 1)

Ein Blazor in drei Geschmacksrichtungen

Das Blazor-Framework bietet aktuell drei Ausführungs- und Hostingmodelle zur Nutzung an [1]: Zum einen gibt es Blazor Server, das sich beim Ausführen der Anwendung auf einen zustandsbehafteten Server verlässt. Das Rendern der Razor-Templates nach HTML findet dabei auf einem Server statt. Über eine SignalR-Verbindung werden Benutzerinteraktionen vom Webbrowser an den Webserver gemeldet, was dort ein erneutes Rendern auslöst. Der Server liefert ein Diff der geänderten HTML-Inhalte über dieselbe Verbindung zurück. Schematisch dargestellt ist dies in Bild 1. Auf dem Server kann jeder beliebige .NET-Code ausgeführt werden und sämtliche plattformspezifischen Schnittstellen lassen sich nutzen; außerdem ist der Code dort vor fremder Einsichtnahme geschützt.



Beim SPA-Modell werden alle Quelldateien zum Client übertragen (Bild 2)

Dem steht Blazor WebAssembly gegenüber, bei dem die Anwendung als Single Page Application (SPA) komplett auf dem Client ausgeführt werden kann. Beim Aufrufen der Anwendung werden alle erforderlichen Quelldateien vom Webserver geladen. Dies kann ein statischer Webserver sein, besondere Anforderungen an den Server gibt es nicht. Zur Laufzeit werden HTTPS- und WebSocket-Verbindungen (etwa via SignalR) genutzt, um Daten auszutauschen (siehe Bild 2).

Der Begriff WebAssembly bezeichnet dabei den gleichnamigen Bytecode für das Web, der durch die JavaScript-Laufzeitumgebungen aller gängigen Browser ausgeführt werden kann. Code aus beliebigen Programmiersprachen kann in diesen Bytecode kompiliert und damit im Webbrowser ausführbar gemacht werden. Voraussetzung dafür ist, dass alle verwendeten Schnittstellen auch im Browser zur Verfügung stehen. Beispielsweise ist aus dem Browser heraus kein wahlfreier Zugriff auf das Dateisystem oder auf Geräteschnittstellen möglich. Bei Blazor WebAssembly werden .NET-Assemblies auf einer nach WebAssembly kompilierten Common Language Runtime (CLR) ausgeführt. Wird die Ahead-of-Time-Kompilierung (AoT) eingesetzt, so wird der C#-Code größtenteils direkt nach WebAssembly kompiliert, was Performancevorteile bringen kann, etwa für besonders performancekritische Anwendungen wie Bildbearbeitung.

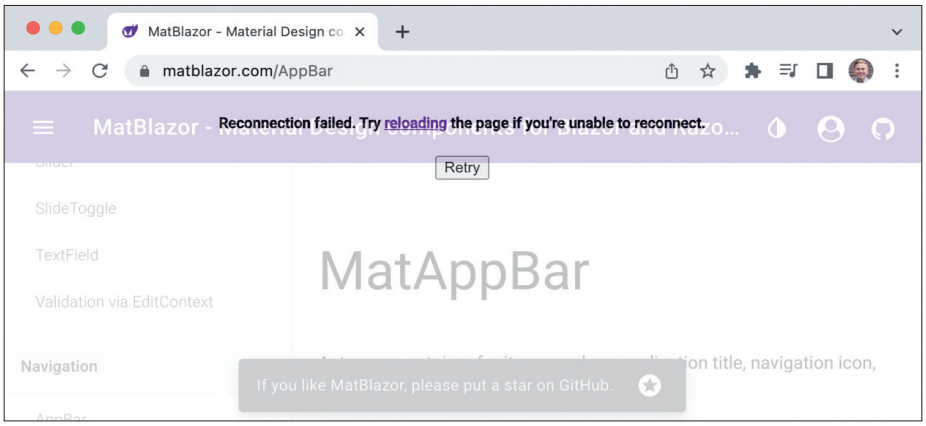
Dann gibt es noch eine dritte Ausprägung: Blazor Hybrid. Hierbei wird die webbasierte Benutzeroberfläche via WebView in eine .NET-MAUI- (Multi-Platform App UI)-, WPF- oder Windows-Forms-Anwendung eingebettet. Das Rendern übernimmt die Anwendung selbst, WebAssembly kommt nicht zum Einsatz. Die Quelldateien werden mit dem Anwendungspaket ausgeliefert, sodass die Anwendung auch offline ausgeführt werden kann. Außerdem lassen sich wie bei Blazor Server sämtliche plattformspezifischen Schnittstellen ansprechen.

Blazor WebAssembly ist oft das Mittel der Wahl

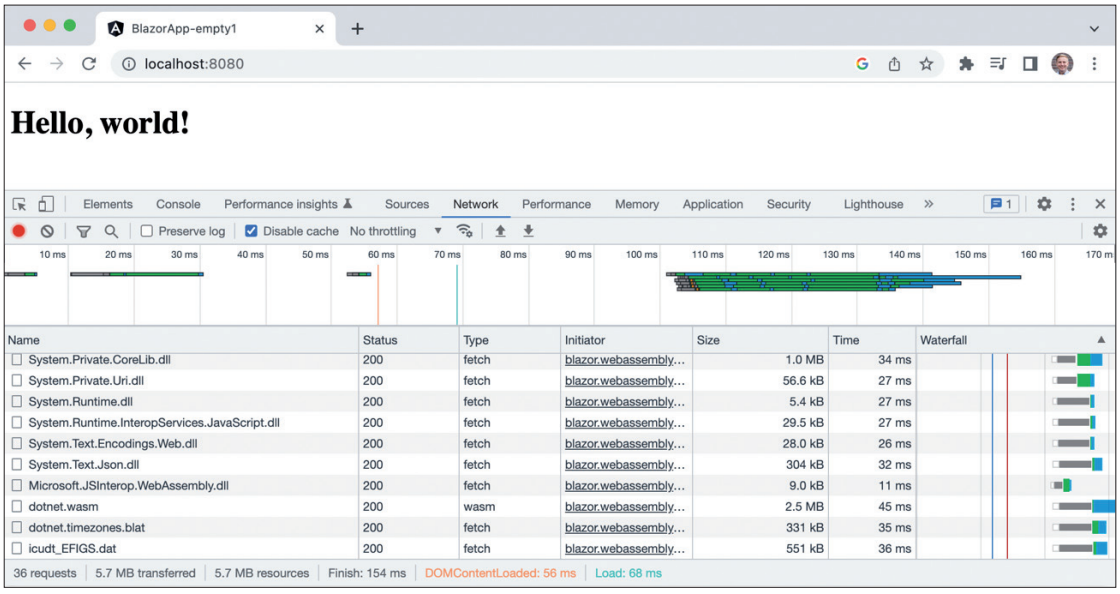
In diesem Artikel soll es vorwiegend um Blazor WebAssembly gehen, das für viele Arten von Webanwendungen als geeigneter erscheint. Denn Blazor Server setzt den Betrieb eines Servers sowie eine dauerhafte und stabile Verbindung mit niedriger Latenz dorthin voraus. Verzögerungen sind schon im Bereich weniger Dutzend Millisekunden für den Anwender spürbar. Reißt die Verbindung ganz ab, lässt sich die Anwendung gar nicht mehr bedienen, wie Bild 3 zeigt. Da der Server zustandsbehaftet ist, kann er nur eine bestimmte Anzahl von Anwendern gleichzeitig bedienen. Für die Skalierung kann dies ein Problem darstellen.

Offline-Szenarien lassen sich mit Blazor Server ebenfalls nicht sinnvoll abbilden, wie es etwa für Progressive Web Apps (PWA) wünschenswert wäre [2]. Mit Blazor Hybrid [3] lassen sich hingegen nur Desktop- oder Mobilanwendungen bauen, was ein Deployment über die üblichen plattformspezifischen Wege wie Installer oder App Stores voraussetzt – im Browser können diese Anwendungen nicht ausgeführt werden. Umgekehrt ließen sich jedoch in Blazor WebAssembly geschriebene Anwendungen, die im Browser laufen, auch für Desktop- und Mobilplattformen verpacken, etwa über Electron oder MAUI. Eine Übersicht über die oben erwähnten Punkte gibt Tabelle 1.

In den Projekten, die wir von Thinktecture begleiten durften, stellte sich Blazor WebAssembly überwiegend als die geeignetere Wahl heraus. Nur in einem Projekt, bei dem es um eine Anlagensteuerung ging, fiel die Wahl auf Blazor Server, da aus dem Webbrowser heraus nicht direkt mit der Maschinenanlage kommuniziert werden kann und der steuernde ▶



Bei einer unterbrochenen Serververbindung ist bei Blazor Server standardmäßig keine Nutzung mehr möglich (Bild 3)



Eine Hello-World-Blazor-Anwendung umfasst bereits 5,7 MByte. Zu sehen sind auch die übertragenen DLL-Dateien (Bild 4)

PC direkt neben der Anlage steht, weswegen Offline-Szenarien nicht relevant sind. Der Ansatz bietet sich außerdem an, wenn der Binärcode aus Geheimhaltungsgründen nicht auf den Rechner des Anwenders übertragen werden soll.

Muss es denn Blazor sein?

Blazor WebAssembly wurde im Mai 2020 erstmals veröffentlicht und gesellte sich zu altgedienten Frontend-Frameworks wie Angular hinzu. Verglichen mit diesen ist Blazor WebAssembly noch immer ein junges Framework mit entsprechendem Wachstumspotenzial. Dank der Verfügbarkeit von AoT-Kompilierung, serverseitigem Prerendering und Native Dependencies sowie der Verbesserung bei der Interoperabilität mit JavaScript im Rahmen der Veröffentlichung von .NET 6 wurde im November 2021 erstmals ein sinnvoller Reifegrad erreicht, sodass wir von Thinktecture den Einsatz des Frameworks – bei passenden Rahmenbedingungen in den jeweiligen Projekten – grundsätzlich empfehlen können. Microsoft verbessert Blazor auch stetig: So kamen mit .NET 7 im November 2022 weitere nützliche Features hinzu, etwa können Blazor-Komponenten jetzt direkt als Web Components verwendet werden; Verbesserungen gab es beim Routing und Data Binding. Einige interessante Features wie Multithreading haben den Einzug in den Release aber leider verpasst.

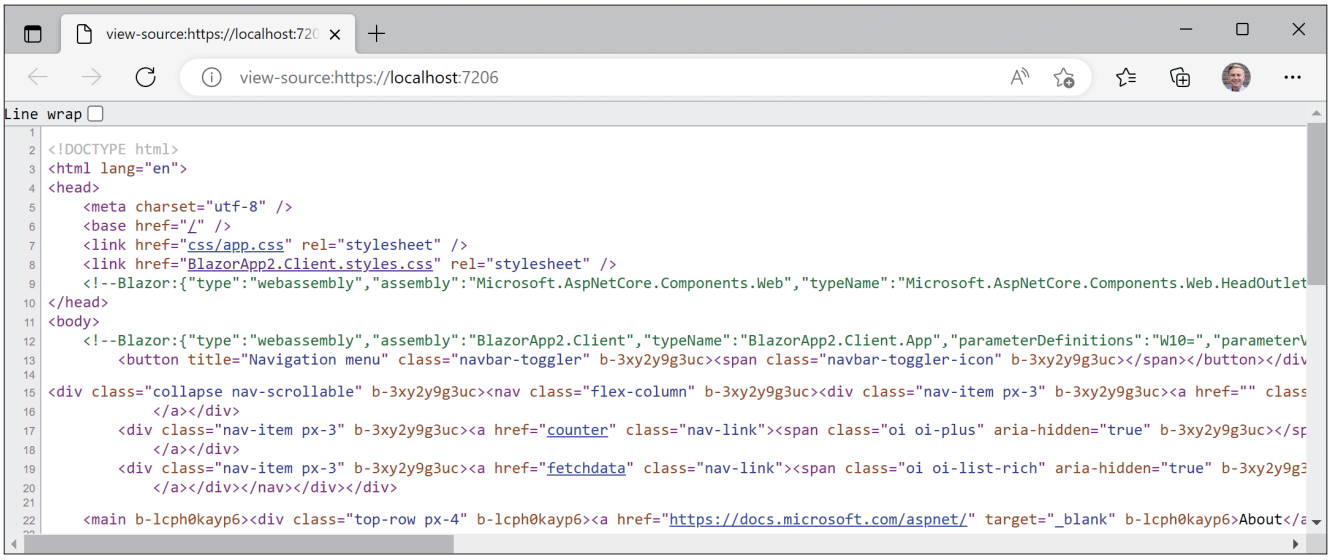
Ganz am Anfang eines Projekts steht oftmals die Wahl einer geeigneten Technologie. Diese ist unter anderem abhängig von den Erfahrungen und dem Entwicklungspotenzial eines Entwicklerteams. Angular nutzt viele Konzepte, die von der Extensible Application Markup Language (XAML) und der Windows Presentation Foundation (WPF) bekannt sind. Die bei Angular verwendete Programmiersprache TypeScript wurde von Anders Hejlsberg entworfen, also demselben Sprachdesigner, der auch für C# verantwortlich zeichnete. Aus diesen Gründen liegt auch Angular einem .NET-Team relativ nahe.

Tabelle 1: Funktionsmerkmale der Hostingmodelle

	Blazor WebAssembly	Blazor Server	Blazor Hybrid
Aufruf sämtlicher plattformspezifischer Schnittstellen	– (nur wenn paketiert)	ja (nur auf Server)	ja
Unterstützung für Offline-Fähigkeit	ja	–	ja
Im Browser ausführbar	ja	ja	–
Paketierbar für Mobil-/Desktop-Plattformen	ja	–	ja
Code kann geheim gehalten werden	–	ja	–

Blazor ist kein „File > New Project > Windows to Web“

Wo es ausschließlich .NET- und C#-Expertise gibt, bietet sich Blazor eventuell eher an. Doch auch bei Blazor kommen Entwickler nicht um die Hypertext Markup Language (HTML), die Cascading Style Sheets (CSS) und am Ende auch JavaScript herum. Nichttriviale Blazor-Projekte werden dennoch mit JavaScript interagieren müssen, insbesondere für die Integration bestehender JavaScript-Bibliotheken. Während die Zahl von Blazor-Komponenten nach wie vor eher überschaubar ist, gibt es eine Vielzahl an JavaScript-basierten Grafikbibliotheken, PDF-Generatoren und Ähnlichem. Was aus Frameworks wie Angular direkt genutzt werden kann, erfordert bei Blazor WebAssembly die umständliche Nutzung der JS-Interop-Brücke und oftmals das Schreiben von passendem Bindungscode in JavaScript [4]. Die JS-Interoperabilität hat sich mit .NET 5 jedoch deutlich verbessert: Es ist nun möglich, JavaScript-Module direkt zu laden, anstatt Objekte auf dem globalen Window-Objekt verfügbar machen zu müssen. Auch die Zugriffe auf JavaScript-Objektreferenzen funktionieren mittlerweile hinreichend schnell.



Bei serverseitigem Prerendering wird das HTML bereits auf dem Server zusammengesetzt, was die gefühlte Ladezeit verkürzt (Bild 5)

Ebenso müssen sich mit Blazor WebAssembly geschriebene Single Page Applications in dieselben Authentifizierungsflows integrieren wie andere Webanwendungen auch. Aktuell basiert die Authentifizierung in Blazor auf der in die Jahre gekommenen oidc-client.js-Bibliothek; ein für .NET 7 geplanter Umbau schaffte es leider nicht in den Release [5].

Auch in Blazor wird das Modell des Component-Based Software Engineering (CBSE) eingesetzt: Die Anwendung wird dabei in viele kleine, wiederverwendbare Komponenten zerlegt. Dahingehend wurde das Framework zuletzt ebenfalls weiterentwickelt, so ist es seit .NET 5 etwa möglich, CSS-Stile komponentenspezifisch zu hinterlegen.

Weiterhin läuft das Framework im Browser; Entwickler müssen auch dort nach Fehlermeldungen schauen und diese verstehen können. Blazor WebAssembly ist also keine wundersame Maschine, die .NET einfach so ins Web übersetzt. Es handelt sich vielmehr um dieselbe Abstraktionsebene, die auch Angular, React oder Vue bedienen – inklusive aller Vor- und Nachteile.

Entwickler sollten zudem Windows-Systeme einsetzen. Auch wenn Blazor-Apps mit JetBrains Rider oder auf Visual Studio for Mac implementiert werden können und die Unterstützung einige Verbesserungen erfuhr, funktioniert die Entwicklung inklusive Debugging nur in Visual Studio 2022 auf Windows richtig gut. So funktioniert etwa Hot Reload, das sofortige Neuladen der Anwendung nach Änderungen im Quelltext, nur in dieser Entwicklungsumgebung sinnvoll. Rider hatte unmittelbar nach Freigabe von .NET 7 noch keine vollständige Unterstützung für das neue SDK.

Mobile Nutzer und Firewalls als Hürden

Weiterhin entscheidend für die Technologiewahl ist die Zielgruppe der Anwendung: Handelt es sich um eine öffentlich erreichbare Anwendung oder um ein rein intern verwendetes Tool? Grund dafür ist die Bundle-Größe von Blazor-WebAssembly-Anwendungen. Selbst eine einfache Hello-World-Anwendung wird mit Blazor WebAssembly schon 5,7 MByte groß (siehe Bild 4), eine vergleichbare Angular-Anwendung beginnt hingegen bei rund 150 kByte Größe. Vergleichsgrundlage hierbei sind die Hello-World-Anwendungen, die per *dotnet new blazorwasm-empty* beziehungsweise *ng new* erzeugt werden. Gezählt werden die übertragenen Bytes der unkomprimierten produktiven Builds, die sich mit *ng build* beziehungsweise *dotnet publish -c release* bauen lassen. Verwendete Versionen waren das .NET-SDK 7.0.100 und Angular 15.0.0. Mit aktivierter Brotli-Kompression lässt sich die Größe bei Angular auf circa 50 kByte reduzieren; schaltet man in Blazor WebAssembly zusätzlich auch noch Kulturinformationen ab, kann man dort 2 MByte Bundlegröße erreichen. Blazor legt die DLL-Dateien außerdem im Cache-Speicher ab, sodass sie beim nächsten Aufruf nicht erneut heruntergeladen werden müssen.

Während es zuletzt deutliche Verbesserungen gab, wird der Größenoverhead bei Blazor jedoch nie ganz verschwinden: Da .NET nicht auf dem Web aufsetzt, müssen die passenden Übersetzungsschichten mitgeliefert werden. Soll die Anwendung auch mobil verwendet werden, kann der schiere Umfang von Blazor-Apps schon der Dealbreaker sein. Auch die initiale Laufzeitperformance von Blazor-Anwendungen ist nicht ideal: Nach dem Hochfahren von Blazor werden die .NET-Assemblies heruntergeladen, somit ergibt sich für den Benutzer eine lange Ladezeit, in der er einen Ladebildschirm sieht.

In einem Projekt verhinderte die bei einem Endkunden eingesetzte Firewall, dass die DLL-Dateien der .NET-Assemblies heruntergeladen wurden und die Blazor-App hochfahren konnte. Es ist nachvollziehbar, dass Security-Lösungen das Herunterladen einer Vielzahl von DLL-Dateien als Bedrohung ansehen. Die AoT-Kompilierung schafft hier leider keine Abhilfe, sondern verschlimmert die Größensituation sogar noch, da die dadurch entstehenden Wasm-Bündel teilweise doppelt so groß werden. Außerdem wird die Anzahl der übertragenen DLLs nur verringert, da etwa für Reflection weiterhin .NET-Assemblies übertragen werden müssen. Ob sich ▶

dies noch weiter optimieren lässt, ist fraglich. Insofern ist Blazor WebAssembly für B2C-Lösungen, bei denen die Umgebung wie etwa bei Mobileinsatz oder Firewalls nicht kontrollierbar ist, eher ungeeignet. Doch selbst für B2B-Anwendungen oder rein interne Lösungen kann die Größe und Laufzeitperformance von Anwendungen problematisch werden: Ein Mitarbeiter eines Kunden, der sich von zu Hause per VPN in das Unternehmensnetz einwählt, muss auf den Start der Blazor-Anwendung rund 20 Sekunden warten. Ein anderer Kunde wechselte aufgrund der im Vergleich deutlich schlechteren Laufzeitperformance frustriert von Blazor WebAssembly zu Angular, da er dieses Framework zuvor bereits eingesetzt hatte und Blazor keinerlei Vorteile bot.

Eingebaute Lösungen für bessere Performance

Teilweise Abhilfe kann hier das serverseitige Prerendering von Blazor WebAssembly schaffen, das seit .NET 5 verfügbar ist. Dabei wird die Seite serverseitig vorgerendert und das fertige HTML an den Client übertragen (siehe Bild 5). Dies wiederum setzt nun aber wieder den Betrieb eines dafür konfigurierten Servers voraus. Ohne Prerendering können die Quelldateien der Anwendung auf jeden beliebigen statischen Webserver übertragen und von dort ausgeführt werden. Außerdem müssen Entwickler das Verhalten der Anwendung im Preloading-Fall zusätzlich testen.

Einmal geladen, kann eine Blazor-WebAssembly-App jedoch annähernd dieselbe Geschwindigkeit erreichen wie alle anderen Webanwendungen auch – mit denselben Fallstricken, die auch für alle anderen Frameworks gelten: etwa gebremst durch das Rendern zu vieler Daten. Ein wichtiger Faktor für die Laufzeitperformance ist die Anzahl der dargestellten Knoten im Document Object Model (DOM). Eine Liste mit Tausenden Zeilen sollte nicht direkt gerendert werden. Alternativen sind Paging oder virtualisierte Listen. Für Letzteres bietet Blazor WebAssembly im Gegensatz zu Angular und Co. sogar eine eingebaute Lösung an: Die Komponente *<Virtualize>* optimiert die Darstellung von Listendaten, indem nur die gerade sichtbaren Datensätze wirklich in das DOM eingefügt werden. Die zugrunde liegenden Daten können entweder im Speicher gehalten oder bei Bedarf dynamisch nachgeladen werden. Somit ergibt sich selbst für sehr lange Listen eine optimale Laufzeitperformance [6]. Als weitere Option kam mit .NET 7 das experimentelle QuickGrid hinzu, das auch Paging beinhaltet [7].

Fazit

Blazor WebAssembly eignet sich vorrangig für interne Enterprise-Anwendungen, bei denen die Umgebung klar kontrolliert werden kann, wo es bestehende .NET- und C#-Expertise gibt, bei denen ein Umstieg auf andere Ansätze nicht sinnvoll realisierbar ist und Windows als Entwicklungsumgebung zum Einsatz kommt.

Das ist allerdings nicht ausschließlich zu verstehen: Einer unserer Kunden verwendet Blazor WebAssembly eben doch für eine B2C-Webanwendung. Die Anwender tolerieren die etwas längere Ladezeit, die immer noch kürzer ausfällt als bei der ein oder anderen Windows-Anwendung, der Kunde kann die App dank vorhandener C#-Kenntnisse weitestgehend selbst entwickeln und holt sich für die JavaScript-Integration und Fehlerbehebungen dann wieder Unterstützung von außen ins Projekt.

Aufgrund seiner ungewöhnlichen Architektur und schlechten Skalierbarkeit empfehlen wir, Blazor Server nach Möglichkeit zu vermeiden und Blazor WebAssembly vorzuziehen. Nur diese Ausprägung des Frameworks passt zu anderen Single-Page-App-Ansätzen wie Angular, React oder Vue, unterstützt Offline-Fähigkeit und lässt sich unter Verwendung derselben Quelldateien auch in Electron oder MAUI hosten.

Kurzum funktioniert das Blazor-WebAssembly-Konzept für Forms-over-Data-Businessanwendungen sehr gut, es ist hinreichend stabil und reif. Zudem entwickelt Microsoft die Framework-Familie beständig weiter. ■

[1] ASP.NET Core Blazor hosting models, www.dotnetpro.de/SL2304BlazorPraxis1

[2] ASP.NET Core Blazor Progressive Web Application (PWA), www.dotnetpro.de/SL2304BlazorPraxis2

[3] ASP.NET Core Blazor Hybrid, www.dotnetpro.de/SL2304BlazorPraxis3

[4] ASP.NET Core Blazor JavaScript interoperability (JS interop), www.dotnetpro.de/SL2304BlazorPraxis4

[5] Secure ASP.NET Core Blazor WebAssembly, www.dotnetpro.de/SL2304BlazorPraxis5

[6] ASP.NET Core Blazor performance best practices, www.dotnetpro.de/SL2304BlazorPraxis6

[7] QuickGrid for Blazor, www.dotnetpro.de/SL2304BlazorPraxis7

Christian Liebel
entwickelt als Softwarearchitekt bei Thinktecture in Karlsruhe Cross-Platform-Apps auf Basis von HTML5 und JavaScript. Sein Handwerk hat er mit Microsoft-Technologien gelernt und er wurde als Microsoft MVP ausgezeichnet.
@christianliebel

Sebastian Gingter
ist Consultant und „Erklärbär" bei Thinktecture. Seine Spezialgebiete sind Backends mit ASP.NET Core, Frontends mit Blazor, Identity und Access Management (IAM), Softwarequalität, alles rund um Tooling und wie man als Entwickler möglichst produktiv sein kann.

Patrick Jahr
ist Softwareentwickler bei der Thinktecture AG. Er unterstützt Kunden bei der Entwicklung und Implementierung moderner Softwarelösungen. Sein Fokus liegt auf modernen (Web-)Technologien, sowohl auf dem Client mit Blazor und Angular als auch mit .NET.

dnpCode	A2304BlazorPraxis