PLATTFORMÜBERGREIFENDE ANWENDUNGEN

Was lange währt

Zwei Jahre nach der ersten Ankündigung veröffentlicht Microsoft .NET MAUI – ein Überblick.

Bild: Klyaksun, Shutterstock

Plattformunabhängige Anwendungen mit einer einheitlichen Nutzerschnittstelle: Das ist ein Traum vieler Entwickler. Möglichkeiten dazu gibt es bereits seit einiger Zeit. Java als Laufzeitumgebung wirbt damit praktisch seit der Erstveröffentlichung der Programmiersprache, des Java Development Kit (JDK) und der Laufzeitumgebung als Softwareplattform. Auch C++ mit entsprechenden Bibliotheken ist eine Option sowie Qt, und mittlerweile kommen noch zahlreiche andere Technologien dazu, wie Electron und Tauri, mit denen in Form eines nativen Containers Webtechnologien wie JavaScript, TypeScript, HTML und CSS auf den Desktop gebracht werden können.

Die Forderung nach plattformunabhängigen Anwendungen ist mit den Jahren immer lauter geworden. Für viele Anwendungsfälle ist es nicht mehr möglich, ausschließlich auf eine Plattform zu setzen. Zudem werden unter dem Plattformbegriff nicht mehr nur Windows, macOS und Linux verstanden, sondern auch mobile Anwendungen. Spätestens seit dem Aufkommen von Android und iOS sind diese beiden Systemumgebungen für viele Anwendungsfälle ebenfalls zu berücksichtigen.

Microsoft hat sich hier bei den ersten .NET-Versionen nicht mit Ruhm bekleckert. .NET war in der Ursprungsform für Windows ausgelegt. Später kamen Webanwendungen hinzu, aber auch hier war die Ausführung lange Zeit auf Microsoft-Technologien beschränkt. Offensichtlich wurde der Druck aber zu groß, sodass beispielsweise mit .NET Core und anderen Technologien schrittweise eine Annäherung erfolgte, um .NET auch auf anderen Systemen als Windows zum Laufen zu bekommen.

Mit .NET MAUI ist nun praktisch der letzte Schritt vollzogen, um mit .NET eine einheitliche Oberfläche für verschiedene Plattformen erstellen zu können. .NET MAUI ist seit Mitte Mai 2022 final veröffentlicht [1], und schon die Release Candidates (RC) erfreuten sich regen Interesses. Angekündigt hatte Microsoft diese Technologie bereits Mitte Mai 2020

im Rahmen von .NET 5 [2], als die Reise hin zu einer vereinheitlichten .NET-Plattform begann, um .NET Core und Mono/Xamarin in einer „Base Class Library" (BCL) und einer entsprechenden Toolchain (SDK) zusammenzuführen. Die Erwartungen an .NET MAUI sind demnach sehr groß. Somit ist es Zeit, einen ersten Einblick in .NET MAUI zu bekommen, als Technologie und als Möglichkeit, plattformunabhängige Anwendungen zu erstellen.

Herausforderung und Chancengeber

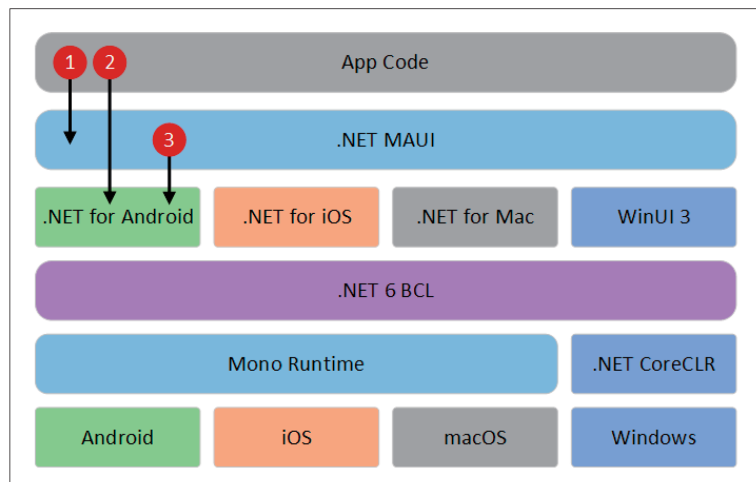
Nicht nur auf eine Plattform beschränkt zu sein ist eine Herausforderung und eine Chance zugleich. Viele Unternehmen wünschen sich daher eine entsprechende Strategie und verfolgen diese zum Teil auch, um ihre Anwendungen auf mehreren Plattformen zur Verfügung zu stellen. Wer das mit Webanwendungen gut machen kann, gewinnt hier bereits enorm viel. Müssen es native Anwendungen sein, werden die Chancen von plattformunabhängigen Anwendungen schnell zu Herausforderungen. Ein und dieselbe Anwendung für mehrere Plattformen zu schreiben ist äußert aufwendig, kostenintensiv und letztendlich fehleranfällig – insbesondere dann, wenn Code reproduziert werden muss.

Eine Plattform komplett zu ignorieren wird inzwischen allerdings immer schwieriger und ist für viele Unternehmen im Jahr 2022 schlichtweg nicht mehr zu vertreten. Kunden verlangen danach, eine Anwendung auf unterschiedlichen Geräten, beispielsweise PC und Smartphones, nutzen zu können, entweder für den eigenen Bedarf oder weil sie Familie, Freunden und Bekannten davon erzählen. Gibt es die Anwendung dann nicht auf der jeweils anderen Plattform, führt das schnell zu einem Imageverlust.

Die Herausforderungen von neu implementierten Anwendungen pro Plattform oder von geteilten Codebasen bleibt aber. .NET MAUI adressiert das auf eine Weise, die viel Aufwand spart und die Produktivität erhöht.

.NET MAUI ist Open Source und die Weiterentwicklung von Xamarin.Forms mit von Grund auf neu entwickelten UI-Steuerelementen. Wer Xamarin.Forms zum Erstellen plattformübergreifender Nutzeroberflächen schon verwendet hat, wird viele Ähnlichkeiten mit .NET MAUI feststellen.

Eines der Hauptziele von .NET MAUI ist es, die Möglichkeit zu bieten, einen möglichst großen Teil der Anwendungslogik und des UI-Layouts in einer einzigen Codebasis zu implementieren. Das System erlaubt es, in einem einzigen Projekt eine plattformübergreifende Anwendung zu erstellen. Bei Bedarf ist es aber möglich, plattformspezifischen Quellcode und andere Ressourcen hinzuzufügen. .NET MAUI ist somit für alle geeignet, die plattformübergreifende Anwendungen in XAML und C# implementieren wollen und dabei möglichst viel Code gemeinsam in beispielsweise Visual Studio nutzen möchten. Zudem ist auch die gemeinsame Nutzung von UI-Layouts und Designs auf verschiedenen Plattformen möglich, ebenso wie das plattformübergreifende Teilen von Code, Tests und Geschäftslogik.



Die Zugriffsmöglichkeiten von Anwendungscode auf .NET über .NET MAUI über die einzelnen Schichten (Bild 1)

Was ist .NET MAUI?

Aber was ist .NET MAUI denn nun konkret? Die Abkürzung steht für „.NET Multi-Platform App UI". Das legt schon im Namen den Fokus auf die Nutzeroberfläche und darauf, dass jetzt mehrere Systeme adressiert werden. Zudem steckt .NET als Begriff im Namen. Es handelt sich somit um ein Framework, das auf .NET aufbaut, mehrere Plattformen adressiert und dabei die Oberfläche im Fokus hat; mit den Plattformen sind native mobile Apps und Desktop-Anwendungen gemeint. Als Technologien kommen zudem C# und XAML zum Einsatz sowie das .NET Framework in Version 6. Oft ist die Rede von .NET MAUI 6 oder .NET MAUI 7, um Bezug auf .NET 6 und .NET 7 zu nehmen. Microsoft scheint diese Namen aber nicht stringent zu verwenden. .NET MAUI erlaubt es, eine einzelne geteilte Codebasis zu verwenden, um Anwendungen für Android, iOS, macOS und Windows zu implementieren.

Wer die Windows Presentation Foundation (WPF) oder WinUI 3 kennt, die ebenfalls XAML-Implementierungen sind, wird festgestellt haben, dass einige Elemente unterschiedlich benannt sind. Ein *TextBox*-Element unter WPF ist jetzt ein *Entry* unter .NET MAUI und *TextBlock* unter WPF ein *Label* in MAUI. In der Konsequenz sorgen diese und weitere Unterschiede dafür, dass WPF- oder WinUI-3-XAML-Code nicht mit .NET MAUI kompatibel ist. Mindestens die Namen der Elemente sind anzupassen.

Der Ansatz und die Architektur von .NET-MAUI-Apps

Als Basis nutzt .NET MAUI das .NET Framework von Microsoft. Das überrascht kaum, ist allerdings nur die halbe Wahrheit. Um .NET auf anderen Systemen nutzen zu können, sind dort angepasste Umgebungen notwendig. Unter .NET sind Software Development Kits (SDKs) für die jeweiligen ▶

Zielplattformen verfügbar. Folgende plattformspezifischen Frameworks gibt es:

- .NET for Android für Apps unter Android
- .NET for iOS für Apps unter iPhone und iPad (iPadOS)
- .NET for Mac für Apps unter macOS
- WinUI 3 für Apps unter Windows

Als Laufzeitumgebung steht unter Windows-Systemen Win32 und auf den anderen Plattformen die Mono-Runtime zur Verfügung. Zudem nutzen alle genannten Plattformen die Klassenbibliothek von .NET 6, um grundlegende Funktionen und insbesondere gemeinsame Datentypen und Datenstrukturen zur Verfügung zu haben. Das ist die bereits erwähnte Base Class Library.

.NET MAUI bietet ein einzelnes Framework, um Nutzeroberflächen für Desktop- und mobile Anwendungen zu erstellen. Bild 1 zeigt die grundlegende Architektur einer .NET-MAUI-App. Der Code interagiert primär mit dem API von .NET MAUI, was der Pfeil (*1*) zeigt. .NET MAUI wiederum greift direkt auf die APIs der nativen Plattform zurück, was schematisch mit Pfeil *3* für .NET für Android angedeutet ist. Zusätzlich ist es möglich, dass App-Code auch direkt auf plattformspezifische APIs zurückgreift (Pfeil *2*).

Android-Apps, die mit .NET MAUI erstellt werden, werden von C# in eine Zwischensprache (IL) kompiliert, die dann beim Starten der App von einem Just-in-Time-Compiler in eine native Assembly übersetzt wird. iOS-Apps, die mit .NET MAUI erstellt werden, werden vollständig „ahead of time" von C# in nativen ARM-Assembly-Code kompiliert. Mit .NET MAUI erstellte macOS-Apps verwenden Mac Catalyst, eine Lösung von Apple, die eine mit UIKit erstellte iOS-App auf den Desktop bringt und sie bei Bedarf mit zusätzlichen AppKit- und Plattform-APIs erweitert. Windows-Anwendungen, die mit .NET MAUI erstellt wurden, verwenden die Bibliothek Windows UI 3 (WinUI 3), um native Anwendungen für den Windows-Desktop zu erstellen.

Voraussetzungen, Installation, Unterstützung

Die Unterstützung von .NET MAUI wird stetig besser. Beim Schreiben dieses Artikels ist der Einsatz zusammen mit Visual Studio noch am besten. Erforderlich ist Visual Studio 2022 in Version 17.3 unter Windows und Version 17.4 unter macOS. Dabei spielt es keine Rolle, ob es die Community-, Professional- oder Enterprise-Variante der IDE ist. Das ist sehr erfreulich, denn damit steht auch denen die Entwicklung mit .NET MAUI offen, die beispielsweise keinen Zugriff auf die kostenpflichtige Professional- oder Enterprise-Edition haben. Wichtig ist, dass das Arbeitspaket (Workload) mit Namen *.NET Multi-Platform App UI-Entwicklung* installiert ist. Entweder nachträglich eingerichtet oder direkt bei der frischen Installation von Visual Studio.

Zum Entwickeln von .NET-MAUI-Anwendungen unter iOS ist zudem ein Mac nötig, der zur aktuellen Version von Xcode kompatibel ist, die aktuelle Version von Xcode selbst sowie eine Apple-ID, die im kostenpflichtigen Apple-Developer-Programm registriert ist. Letzteres ist spätestens bei der Veröffentlichung der Anwendung in Apples App Store notwendig. Zum Debuggen einer Anwendung von Windows auf einem iOS-Gerät reicht die Apple-ID im Apple-Developer-Programm aus.

Die Unterstützung in anderen Entwicklungsumgebungen ist noch ausbaufähig. So unterstützt JetBrains Rider .NET MAUI in einer ersten Version ebenfalls in Rider 2022.2. Diese Rider-Version enthält aber lediglich eine frühe Vorschau auf die .NET-MAUI-Unterstützung. Mit dieser Rider-Version sind Projekte für die Plattformen Android und iOS möglich, ebenso Laufzeit-Konfigurationen, um ein Projekt auszuführen und zu debuggen sowie um eine Anwendung auf einem Zielgerät bereitzustellen. Die Plattformen macOS, Mac Catalyst, Blazor Hybrid und WinUI werden noch nicht vollständig unterstützt.

Wer nach dem Update von Visual Studio 2022 keine Projektvorlagen für .NET MAUI findet, kann die Installation der Workloads über die .NET-Kommandozeile versuchen:

```
dotnet workload install maui
```

In etlichen Fällen scheint das Abhilfe zu schaffen. Wenn dabei oder beim Erstellen eines Projekts in Visual Studio merkwürdige und oft wenig hilfreiche Fehlermeldungen erscheinen, dann ist es sinnvoll, die installierten Versionen von .NET zu prüfen. Beispielsweise sorgte eine Preview-Version von .NET 7 nachweislich für Probleme beim Erstellen von .NET-MAUI-Apps unter Windows 10. Diese und ähnliche Konstellationen sind häufig anzutreffen, wenn Preview-Versionen von .NET, Visual Studio oder anderen Komponenten installiert sind beziehungsweise waren.

Der aktuelle Funktionsumfang

Da .NET MAUI gerade (Anfang Dezember 2022) frisch in der finalen Version erschienen ist, wird sich der Funktionsumfang in zukünftigen Versionen sicherlich noch stark verändern. Das ist anzunehmen, da das Framework derzeit zahlreiche Komponenten enthält, die sich zu einem Großteil auf plattformübergreifende Funktionalitäten der Nutzeroberfläche konzentrieren. Zu diesen Komponenten gehören Layouts, Pages, Cells, Views, Klassen für die Erkennung von Touch-Gesten und Gestensteuerung (*GestureRecognizer*), eine Shell als Navigationsframework, eine Komponente, um Nachrichten zu aggregieren (*MessagingCenter*), und Elemente für den Umgang mit dem Lebenszyklus einer App.

Zur Laufzeit einer .NET-MAUI-App setzt das Framework die Steuerelemente der Zielplattformen ein, während die Beschreibung dieser Komponenten durch allgemeingültige Komponenten in plattformübergreifendem Code umgesetzt ist. Das ermöglicht eine allgemeine Beschreibung einer Komponente, ohne plattformspezifische Besonderheiten zu kennen. Ein Steuerelement wird dabei einer der eben angesprochenen vier Gruppen Views, Layouts, Pages und Cells zugeordnet. Steuerelemente in diesen Gruppen haben sehr unterschiedliche Aufgaben. Mit Pages sind in der Regel bildschirmfüllende Elemente gemeint, beispielsweise für anderen Inhalt, für die Navigation oder für Karteikartenseiten. Cells umfasst Elemente, die beschreiben, wie der Zellenin-

Plattformunabhängigkeit geht auch anders

Neben .NET MAUI gibt es weitere Systeme und Möglichkeiten, um Anwendungen plattformunabhängig für den Desktop und teilweise auch für mobile Geräte zu implementieren. Bekanntheit hat das Electron-Framework erlangt, mit dem in JavaScript oder TypeScript geschriebene Webanwendungen in einem nativen Container auf den Desktop gebracht werden können [10]. Einen ähnlichen Weg schlägt Tauri ein, ein Framework zum Erstellen winziger und sehr schneller Binärdateien für alle wichtigen Desktop-Plattformen [11]. Für die Nutzeroberfläche kann jedes beliebige Frontend-Framework zum Einsatz kommen, das in HTML, JS und CSS kompiliert. Das Backend der Anwendung ist eine Binärdatei auf Basis von Rust mit einem API, mit dem das Frontend interagieren kann.

Ein weiterer Vertreter und sehr bekannt ist Flutter. Dieses Open-Source-Framework von Google dient zum Erstellen von nativ kompilierten und plattformübergreifenden Anwendungen aus einer einzigen Codebasis. Zudem unterstützt Flutter derzeit mehr Plattformen als alle anderen genannten plattformübergreifenden Frameworks, da beispielsweise auch eingebettete Systeme adressiert werden können.

Das ist lediglich ein kleiner Ausschnitt aus den Möglichkeiten, systemübergreifende Anwendungen zu erstellen. Unter .NET ist noch das Avalonia-Framework bekannt und beliebt [12]. Wer gar nicht auf .NET und/oder C# setzen möchte, findet bei Qt [13] und anderen Frameworks ebenfalls viele Möglichkeiten, um nicht auf nur eine Plattform beschränkt zu sein.

halt einer Liste auszusehen hat; sie dienen daher eher als Vorlage für ein visuelles Element, das den Inhalt beschreibt. Layouts sind Layoutcontainer, die sich um die Positionierung von anderen Elementen kümmern. Und zu Views schließlich gehören Elemente auf einer Oberfläche, die klassischerweise gezeichnet und visuell dargestellt werden; dazu gehören Eingabefelder, Listen, Schaltflächen und dergleichen.

Wird .NET MAUI erwähnt, dann ist häufig die Rede von den zahlreichen Steuerelementen, die das Framework zur Verfügung stellt, beispielsweise um Daten anzuzeigen oder auszuwählen, Aktivitäten anzuzeigen, Aktionen anzustoßen und viele weitere. Was .NET MAUI aber zusätzlich bietet, sind plattformübergreifende APIs für Gerätefunktionen, wie der Zugriff auf Sensoren, etwa Beschleunigungsmesser, Kompass und Gyroskop. Zudem lässt sich über eine .NET-MAUI-App der Status der Netzwerkverbindung des Geräts prüfen sowie Änderungen erkennen. Ebenso möglich ist das sichere Speichern von Daten als Schlüssel-Wert-Paare, der Einsatz der integrierten Text-to-Speech-Engines und das Anstoßen von browserbasierten Authentifizierungsabläufen, die auf einen Rückruf zu einem bestimmten, für die Anwendung registrierten URL warten.

.NET MAUI versus Xamarin.Forms

Das neue .NET MAUI wird immer wieder mit dem bereits vorhandenen Xamarin.Forms verglichen. Dieser Vergleich ist nachvollziehbar, denn Xamarin wird seit geraumer Zeit in den verschiedensten Projekten verwendet. Da ist die Neugierde groß, was .NET MAUI kann und besser macht.

Ein Hauptunterschied ist, dass .NET MAUI als deutlich einsteigerfreundlicher und einfacher gilt. Die Zeiten von unterschiedlichen Projekten für die jeweiligen Zielplattformen und eines für den geteilten Quellcode sind vorbei. Dabei stellte sich dem Entwickler immer die Frage, welches Projekt jetzt gerade das korrekte ist, um etwas hinzuzufügen oder zu verändern, insbesondere beim Einstieg in das Thema. Darüber hinaus unterstützt .NET MAUI nicht nur XAML-Hot-Reload, sondern auch .NET-Hot-Reload. Somit lässt sich sowohl XAML-Code als auch der C#-Code zur Laufzeit verändern, und die Auswirkungen zeigen sich direkt in der laufenden Anwendung.

Beim Darstellen der Komponenten wurde der alte Code der Renderer, die Reflexion zum Aufbauen der plattformspezifischen Steuerelemente verwenden, durch neue Implementierungen ersetzt. Diese nutzen jetzt sogenannte Handler, die auf Interfaces und Dependency Injection setzen, um das Steuerelement auf den Bildschirm zu zaubern. Das ermöglicht die Erweiterbarkeit von .NET MAUI und ist zudem schlicht schneller. Des Weiteren setzt Microsoft beim Thema Dependency Injection auf den Namensraum *Microsoft.Extensions*, was deutlich leistungsfähiger und das Dependency-Injection-Framework unter ASP.NET Core ist.

Abschließend sind die Unterstützung von macOS für Desktop-Anwendungen und die Nutzung von Blazor zwei wesentliche Unterschiede. Bei Xamarin.Forms war macOS offiziell gar kein Thema, und die Nutzung von Blazor für hybride Apps ermöglicht den einfachen Einstieg für alle, die eher mit der Webentwicklung vertraut sind.

Das sind zahlreiche Argumente auf der Seite von .NET MAUI. Wie diese in der Community ankommen, ist eine andere Frage. Die Liste der Merkmale bietet aber eine gute Grundvoraussetzung für eine schnelle Adaption von MAUI. Zudem ist zu berücksichtigen, dass die Weiterentwicklung von Xamarin.Forms zugunsten von .NET MAUI eingestellt wurde. Das macht .NET MAUI zum mehr oder weniger offiziellen Nachfolger.

Wer .NET MAUI nicht einsetzen möchte, wird sich über kurz oder lang gänzlich anders orientieren müssen. Auf welche anderen Möglichkeiten man ausweichen kann, zeigt der Kasten Plattformunabhängigkeit geht auch anders.

Die Migration von Apps mit Xamarin.Forms zu .NET MAUI ist ebenfalls möglich. Eine automatische Migration soll früher oder später mit dem .NET Upgrade Assistant funktionieren [3]. Im Moment scheint das aber eher später als früher der Fall zu sein, da der Assistent die automatische Migration von Xamarin.Forms zu .NET MAUI noch nicht beherrscht oder ▶

besser gesagt noch nicht vollständig unterstützt. Die manuelle Migration, sowohl in der Dokumentation von Microsoft [4] als auch im Repository auf GitHub beschrieben [5], funktioniert ebenfalls. In der Dokumentation ist das mit wenigen Schritten dargelegt, die aber im Detail viel Aufwand erzeugen können, beispielsweise das Aktualisieren der Namensräume, ein Update der inkompatiblen NuGet-Pakete und das Beheben der Breaking Changes beim API.

Das schreibt sich hier so einfach, kann im Detail aber ein großes Problem werden, insbesondere dann, wenn beispielsweise für externe Bibliotheken noch gar kein Update für .NET MAUI zur Verfügung steht. Daher ist der Wechsel auf .NET MAUI im Vorfeld gut darauf zu prüfen, ob er technisch überhaupt machbar ist.

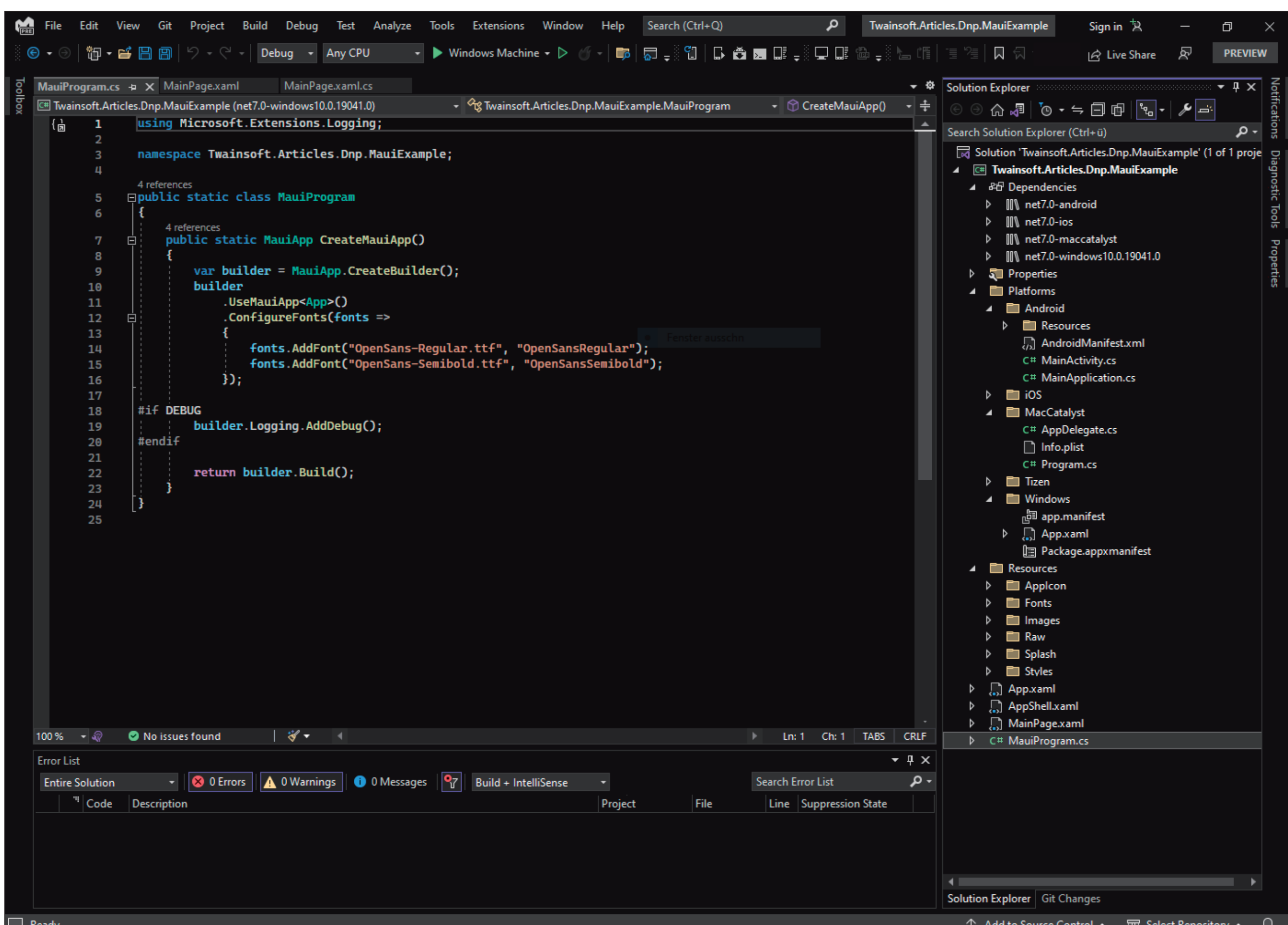
Die Struktur einer .NET-MAUI-App

Nachdem alles korrekt auf dem Entwickler-PC eingerichtet ist, lässt sich über die .NET-MAUI-Vorlage für eine MAUI-App ein neues Projekt in Visual Studio erzeugen. Da JetBrains Rider noch nicht gut für .NET MAUI gerüstet ist, kommt für diese Demo Visual Studio 2022 in Version 17.5.0 Preview 1.0 zum Einsatz. Basis ist C# 11 und das .NET Framework in Version 7.0.100.

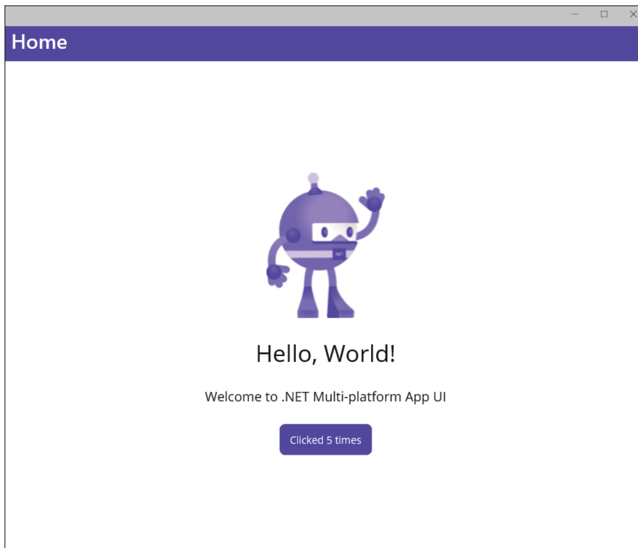
Solange Sie keine leere Projektmappe erzeugen, generiert Visual Studio bereits etliche Elemente in Form von Verzeichnissen, Quelltextdateien und Ressourcen. Das liegt am Democharakter eines neuen Projekts, aber insbesondere an den unterschiedlichen Anforderungen der Plattformen. Generiert werden nämlich sowohl plattformübergreifender Quellcode und notwendige Ressourcen als auch plattformspezifischer Code und Ressourcen. Letztere befinden sich in eigenen Verzeichnissen, passend zu den jeweiligen Plattformen. Bild 2 zeigt einen Screenshot von Visual Studio 2022 mit zum Teil aufgeklappter Projektstruktur.

Beim Erzeugen des Projekts wird der plattformübergreifende Quellcode gemeinsam mit dem plattformspezifischen Quellcode in die Anwendung kompiliert. Die Elemente in den plattformspezifischen Unterordnern, die nicht zur ausgewählten Zielplattform passen, werden dabei ignoriert. Auf diese Weise lassen sich plattformspezifische Anwendungen erstellen und man kann trotzdem eine einzelne Projektstruktur verwenden. Letzteres macht das Handling bei der Entwicklung erheblich einfacher.

Die jeweiligen Unterordner für die einzelnen Plattformen verdeutlichen dabei, wie dort beispielsweise der Einstiegspunkt in die Anwendung funktioniert. Bei Android sind das die Klassen *MainApplication* und *MainActivity* und bei Mac Catalyst ist es die *Main()*-Methode in der Datei/Klasse *Program.cs*. Dort wird der Message-Loop über einen *AppDelegate* gestartet.



Ein .NET-MAUI-Projekt, das in Visual Studio 2022 geöffnet wurde (Bild 2)



Die gestartete erste Anwendung unter Windows 10 (Bild 3)

Zusätzlich wird das Verzeichnis *Resources* erzeugt, das etliche Unterverzeichnisse enthält. Beispielsweise für das Anwendungssystem der späteren Anwendung (*Resources\AppIcon*), für die unterschiedlichen Schriftarten (*Resources\Fonts*), für Bilder (*Resources\Images*), Stile für den Einsatz von CSS bei Webanwendungen (*Resources\Styles*), Splash-Screens (*Resources\Splash*) und für statische Dateien, die mit der Anwendung ausgeliefert werden sollen (*Resources\Raw*). Schließlich liegen im Root-Verzeichnis des Projekts zahlreiche Dateien, um die Anwendung ein erstes Mal zu starten und um die initiale Inhaltsseite (*MainPage*) anzuzeigen. In Bild 3 ist diese *MainPage* des zum ersten Mal erzeugten Projekts zu sehen.

Layouts, Formulare und Navigation

.NET MAUI bietet zahlreiche Elemente an, um Anwendungen mit Framework konzipieren und implementieren zu können. Dazu gehören zum Beispiel Layoutcontainer, die für das Oberflächendesign zuständig sind. Die Grundprinzipien sind von anderen XAML-basierten Layoutsystemen bekannt, zum Beispiel bezogen auf die Klassen *StackLayout*, *Grid*, *AbsoluteLayout*, *FlexLayout* und *ScrollView*. Sie ordnen die Elemente einem vorgegebenen Schema entsprechend an und vereinfachen damit die Positionierung von Elementen auf einer Seite der Anwendung.

Bei der Ausgestaltung dieser Seiten gibt es eine Menge grundlegender Steuerelemente. Beispielsweise *Label*, *Progressbar*, *Border* und *Shadow*. Auch *Shape*-Elemente sind vorhanden, mehr- und einzeilige Texteingaben über *Entry* und *Editor*, Schaltflächen, *RadioButton*, *CheckBox*, *Picker* für Dateien und dergleichen. Diese lassen sich in XAML implementieren, über die Layouts platzieren und über den Code-behind ansteuern. Das ist auch über Entwurfsmuster für das Model-View-ViewModel (MVVM) möglich, um auch bei komplexen Anwendungen nicht den Überblick zu verlieren. Dazu gleich mehr weiter unten.

Für die Navigation ist in .NET MAUI die Shell zuständig. Damit lassen sich verschiedene Arten von Navigationen implementieren wie zum Beispiel über Registerkarten, hierarchische Navigation oder Seitenleisten. Bei der Navigation lassen sich Bereiche wie die Kopf- und Fußzeile definieren, um beispielsweise eine seitliche Navigationsleiste aufzuwerten. Zudem bietet .NET MAUI eine routenbasierte Navigation an, wie man sie von Webanwendungen kennt. Sind diese Routen im Code definiert, lassen sich neue Bereiche der Anwendungen über diese Routen ansprechen.

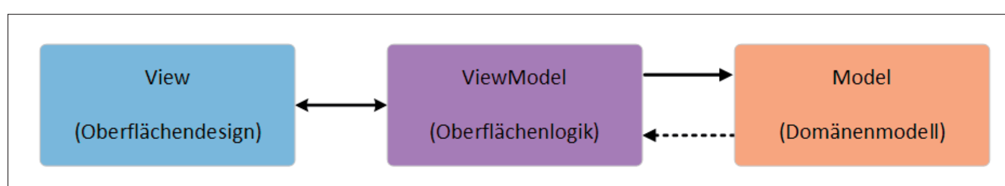
Schließlich runden Listen die Sammlung von Steuerelementen in .NET MAUI ab, um auch komplexe Anwendungen erstellen zu können. Verfügbar ist sowohl *ListView* als auch *CollectionView*. Letzteres gilt als leistungsfähigere und flexiblere Alternative zu *ListView*, mit der sich Listen in zahlreichen Varianten auf dem Bildschirm erzeugen lassen, da diese Komponente beispielsweise auch *DataTemplate*- und *ItemTemplate*-Objekte unterstützt. So bieten Spalten, Zeilen und Zelleninhalte größtmögliche Flexibilität.

Es ist auch möglich, Bilder, Icons und Schriftarten in einer Anwendung zu nutzen, ebenso wie Stile und visuelle Themen. In Kombination lassen sich damit .NET-MAUI-Apps sehr komfortabel optisch anpassen.

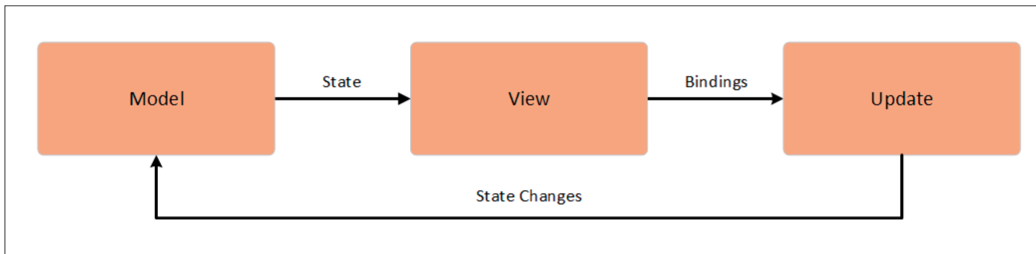
MVVM und/oder MVU?

.NET MAUI setzt wie viele andere auf das Muster MVVM, um Code der Geschäftslogik und der Oberfläche miteinander zu verbinden. Die Nutzeroberfläche wird in XAML deklariert. Um auf diese Elemente zuzugreifen, beispielsweise für einen Klick auf eine Schaltfläche oder um den Text aus einer Eingabe auszulesen, lässt sich dieser Code direkt im Code-behind der Oberfläche einbinden. Was für kleinere Anwendungen sicherlich in Ordnung ist und ausreicht, wird bei umfangreichen Anwendungen unübersichtlich, unleserlich, schwer zu warten und fehleranfällig.

In XAML-basierten Frameworks, zu denen auch Xamarin. Forms oder die Windows Presentation Foundation (WPF) gehören, kommt daher häufig das MVVM-Entwurfsmuster zum Einsatz. Ziel dieses Musters ist es, die Oberflächendarstellung und die Oberflächenlogik voneinander zu trennen und auf diese Art und Weise zu entkoppeln. Das reduziert die oben genannten Probleme erheblich und führt zu besser ▶



Das Entwurfsmuster Model–View–ViewModel (MVVM) als schematische Darstellung (Bild 4)



Die schematische Darstellung des Musters Model-View-Update, kurz MVU (Bild 5)

wartbaren Anwendungen. Die View enthält dabei das angesprochene Oberflächendesign in XAML und den notwendigen Code-behind, das ViewModel die Oberflächenlogik und das Model das Domänenmodell (Bild 4).

Im Zusammenhang mit .NET MAUI ist zudem häufig von dem Entwurfsmuster Model-View-Update (MVU) die Rede. MVU wird auch als die Elm-Architektur bezeichnet, da das Muster den Ursprung in der Community der funktionalen Entwicklung hat und Elm eine funktionale Programmiersprache ist. Die Elm-Architektur ist ein Muster für das Entwickeln interaktiver Anwendungen wie beispielsweise Webanwendungen. Diese Architektur scheint sich in Elm eher natürlich herauszubilden, statt dass sie jemand aktiv erfunden hätte; schon früh haben Elm-Entwickler immer wieder dieselben Grundmuster in ihrem Code gefunden, ohne dieses Muster im Voraus aktiv zu planen und anzustreben. Bei diesem Muster gibt es ebenfalls ein Model, eine View und einen Mechanismus, um über Events die View zu aktualisieren, beispielsweise wenn neue Daten vorhanden sind oder vorhandene Daten verändert werden. Bild 5 zeigt dazu ein schematisches Beispiel und verdeutlicht auch die Abweichungen zum MVVM-Muster in Bild 4.

In der .NET-Welt wird dabei häufig auf das Comet-Toolkit verwiesen [6]. Comet ist ein moderner Weg, um plattformübergreifende UIs zu schreiben. Basierend auf .NET MAUI, folgt Comet dem MVU-Muster und übernimmt die Datenbindung auf fast magische Weise für den Entwickler. Kurzgefasst gibt es in der View eine Methode *Body()* und einen Status, repräsentiert durch verschiedene primitive oder komplexe Datentypen. Über *Body()* werden Status und View verknüpft. Ändert sich jetzt der Zustand, ändert sich automatisch die View. Dieses Muster ist nicht nur angelehnt an Architekturen wie bei Elm, sondern in ähnlicher Weise auch bekannt durch Web-Frameworks wie React und ähnliche.

Comet wurde auf der Grundlage von .NET-MAUI-Handlern entwickelt und bietet seine eigene Implementierung für Schnittstellen wie *Microsoft.Maui.IButton* und andere Steuerelemente an. Jede Plattform, die von .NET MAUI unterstützt wird, kann angesprochen werden. Dazu gehören Windows, Android, iOS, macOS und Blazor; Nicht-MAUI-Anwendungsmodelle, wie zum Beispiel UWP oder WPF, werden nicht unterstützt.

Im Moment befindet sich Comet in einem frühen Entwicklungsstadium in Form eines Proof of Concept und ist noch nicht für die produktive Entwicklung von Anwendungen freigegeben. Auf Basis der General-Availability-(GA-)Version von .NET MAUI wurde eine Version von Comet veröffentlicht, sodass es möglich ist, das Zusammenspiel beider Versionen zu testen.

Zugriff auf Gerätefunktionen

Ein elementarer Punkt bei der Entwicklung plattformunabhängiger Anwendungen ist der Zugriff auf plattformspezifische Merkmale, etwa auf Sensoren eines Smartphones. Eine App, die Zugriff auf die Kamera oder den Lagesensor benötigt, möchte sowohl auf Android als auch auf iOS darauf zugreifen können, um nur zwei Beispiele für Gerätefunktionen zu nennen. Unter Xamarin.Forms hatte sich zu diesem Zweck die Open-Source-Bibliothek Xamarin.Essentials herauskristallisiert, die von Microsoft bereitgestellt wird [7]. Diese Möglichkeiten stehen auch unter .NET MAUI zur Verfügung, sind aber in unterschiedliche Namensräume aufteilt.

Beispielsweise gibt es Klassen für den Zugriff auf Gerätesensoren. Dazu gehören die Klassen *Accelerometer*, *Barometer* und *Magnetometer*, um Zugriff auf den Beschleunigungssensor, den Sensor für den Luftdruck und die Werte für das Magnetfeld zu erhalten. Diese und weitere Klassen befinden sich im Namensraum *Microsoft.Maui.Devices.Sensors*.

Zusätzlich gibt es Plattformfunktionen, mit denen sich jeweils spezifische Anwendungen oder Funktionen erreichen lassen. Dazu gehören unter anderem die Klassen *Clipboard*, *Contacts* und *FlashLight*, um beispielsweise Zugriff auf die Zwischenablage, die Kontakte des Nutzers und die Taschenlampe des Smartphones zu haben.

Darüber hinaus gibt es Klassen für den Zugriff auf App- und Geräteinformationen, wie zum Beispiel die Klassen *Battery*, *Connectivity* und *DeviceInfo*. Damit lassen sich die Daten zum Akku, der Netzwerkverbindung und des Geräts abfragen. Weitere Hilfsfunktionen erzeugen beispielsweise Screenshots (über die Klasse *Screenshot*) und rechnen verschiedenen Einheiten um (Klasse *UnitConverters*).

Kritische Stimmen

Die Meinungen zu .NET MAUI gehen in der Community auseinander. Oft ist das Argument zu hören, .NET MAUI sei langsam, mit Bugs durchsetzt und generell nicht für den produktiven Einsatz geeignet. Diese Argumente gehen auf sehr unterschiedliche Begründungen zurück. Beispielsweise ist es ein Problem, dass Microsoft zwar .NET MAUI entwickelt, aber nichts Eigenes damit auf die Beine stellt. Auch die zahlreichen Workarounds, die notwendig sind, um mit .NET MAUI eine produktive Anwendung zu entwickeln, gelten als Beleg für die zahlreichen Bugs. Oft wird der Vergleich mit Xamarin.Forms aus den Jahren 2014 bis 2015 gezogen, in denen

das Framework lediglich für allgemeine Proof-of-Concept-Implementierungen genügt hat. Zudem ist mit Blick auf das Laufzeitverhalten die Rede davon, dass das Framework durchaus noch weitere ein bis zwei Jahre Entwicklungszeit für den produktiven Einsatz benötige.

Solche und ähnliche Diskussionen gibt es zahlreich im Internet. Positive wie negative Kommentare sammeln sich unter den Blog-Posts von Microsoft und in den Diskussionen bei Redet, Hackernews und Co. Während die einen den ungenügenden Einsatz von Microsoft hinsichtlich .NET MAUI bemängeln, loben andere Microsoft gerade dafür, .NET MAUI überhaupt angegangen und über Jahre entwickelt zu haben.

Ein Framework wie .NET MAUI auf die Beine zu stellen und zu veröffentlichen ist sicherlich ein Kraftakt an sich. Es jetzt auf einen Stand zu bringen, wie ihn Xamarin.Forms heute bereits hat, bedeutet die nächste Anstrengung. Und die Weiterentwicklung ist ebenfalls enorm wichtig. Die Issues auf GitHub im Repository zu .NET MAUI zeigen Anfang Dezember 2022 etwas mehr als 2000 offene und über 4300 geschlossene Einträge an [8]. Ende September 2022 waren es noch etwas mehr als 1700 offene und knapp 3800 geschlossene Issues. Hier passiert also eine Menge auf beiden Seiten. Zudem gibt es auf GitHub im gleichen Repository eine gute Übersicht der Roadmap zu .NET MAUI [9], sowohl für .NET 7 als auch für .NET 8. Einige Erweiterungen beziehen sich beispielsweise auf Steuerelemente für Karten, den Betrieb von mehreren Monitoren sowie die Größe und Startzeit von Anwendungen.

Die Vorteile von .NET MAUI sind gegeben. Dazu gehören unter anderem die vereinfachte Entwicklungserfahrung über die .NET-Kommandozeile, moderne Entwicklungsmuster, die Unterstützung von Hot Reload und die Vereinheitlichung von Bibliotheken. Insbesondere mit dem MVU-Muster kann sich noch einiges ändern, was die Komplexität von Projekten betrifft. Ob .NET MAUI für das eigene Projekt bereits ausgereift genug ist, lässt sich schwer verallgemeinernd beantworten. Wer kein Risiko eingehen möchte, muss sicherlich eine Art Proof of Concept oder Prototyp entwickeln, um kritische Punkte eigener Anwendungen zu testen. Dieser Punkt wird sich aber mit der Zeit aller Erfahrung nach von allein verbessern, wenn sich .NET MAUI weiterentwickelt.

Bei allen positiven sowie negativen Betrachtungen sollte klar sein, dass .NET MAUI als erste finale Version fertig und veröffentlicht ist. Ecken und Kanten wird es an der einen oder anderen Stelle sicherlich geben. Wer vorher viel mit Xamarin.Forms entwickelt hat, sollte sich darauf einstellen, dass noch nicht alle Bibliotheken auf .NET MAUI zur Verfügung stehen. Das betrifft sowohl Open-Source- als auch kommerzielle Projekte. Für aktive Repositories und Projekte ist das aber nur eine Frage der Zeit.

Fazit

Mit .NET MAUI erfüllt sich für viele Entwicklerinnen und Entwickler ein lang gehegter Wunsch nach einem direkt von Microsoft entwickelten, gepflegten und auf aktuellen Standards basierenden Framework für das Erstellen von plattformunabhängigen .NET-Anwendungen mit C# und XAML. Insbesondere aus Sicht von Projekten für mobile Anwendungen bietet .NET MAUI große Versprechungen für die Zukunft. Einsteigerfreundlicher und leistungsfähiger als Xamarin.Forms soll es zudem auch noch sein.

Ob sich .NET MAUI auf breiter Basis durchsetzt, muss die Zeit zeigen. Projekte werden sicherlich nur dann von Xamarin.Forms zu .NET MAUI migriert, wenn es einen sehr guten und nicht wegzudiskutierenden Grund gibt, der plötzlich auf der Roadmap des eigenen Projekts auftaucht – die Unterstützung von macOS als Desktop-Betriebssystem zum Beispiel oder die versprochenen Leistungsgewinne von .NET MAUI.

Ansonsten ist das System für die Neuentwicklung sicher eine Überlegung wert, vor allem, wenn Apps für mobile Endgeräte notwendig sind. Wie immer gilt es, einen Blick auf die benötigten Komponenten eines Projekts zu werfen, bevor eine Neuentwicklung oder eine Migration gestartet wird. Das gilt sowohl für eigene als auch für Komponenten für Drittanbieter. Ansonsten kann es eine böse Überraschung geben, wenn etwas nicht mit oder unter .NET MAUI funktioniert.

Microsoft hat mit .NET MAUI einen sehr soliden Grundstein für ein plattformunabhängiges Entwickeln von Anwendungen mit C# und XAML gelegt. Wie es weitergeht und wie gut das Framework ankommt, wird sich jetzt zeigen müssen. Einen Blick ist MAUI aber sicherlich wert. ■

[1] David Ortinau, Introducing .NET MAUI – One Codebase, Many Platforms, www.dotnetpro.de/SL2302MAUI1

[2] Scott Hunter, Introducing .NET Multi-platform App UI, www.dotnetpro.de/SL2302MAUI2

[3] Übersicht über den .NET-Upgrade-Assistenten, www.dotnetpro.de/SL2302MAUI3

[4] Migrieren Ihrer App von Xamarin.Forms, www.dotnetpro.de/SL2302MAUI4

[5] Migrating from Xamarin.Forms (Preview), www.dotnetpro.de/SL2302MAUI5

[6] Comet, www.dotnetpro.de/SL2302MAUI6

[7] Xamarin.Essentials, www.dotnetpro.de/SL2302MAUI7

[8] Issues im Repository zu .NET MAUI auf GitHub, www.dotnetpro.de/SL2302MAUI8

[9] .NET MAUI Roadmap, www.dotnetpro.de/SL2302MAUI9

[10] Electron, www.electronjs.org

[11] Tauri, https://tauri.app

[12] Avalonia UI, https://avaloniaui.net

[13] Qt, www.qt.io

Dr. Fabian Deitelhoff
arbeitet nach seiner Promotion zum Thema „Source Code Comprehension“ als Tech-Lead Domestics an Cloud-Themen bei Miele. Auch ist er als Autor, Dozent und Softwareentwickler im .NET- und Web-Umfeld tätig.
@FDeitelhoff

dnpCode A2302MAUI