PLATTFORMÜBERGREIFENDES ENTWICKELN

Eine Partie Uno

Wer Apps für mehrere Systeme entwickeln will, findet in dem Framework Uno ein mächtiges Werkzeug.

Ein alter Kalauer besagt, dass die freie Marktwirtschaft oft die besten Lösungen für technische Probleme hervorbringe. Im Fall der plattformübergreifenden Umgebung Uno ist dies definitiv so: Das Projekt entstand einst, als das kanadische Softwareunternehmen Nventive zwar umfangreiche Kenntnisse im Bereich der Microsoft-Technologien hatte, aber mit Android und iOS nur wenig anzufangen wusste.

Heute ist das komplett quelloffene Produkt eine der leistungsstärksten plattformübergreifenden Umgebungen, die derzeit am Markt zur Verfügung stehen. Neben Unterstützung für Linux, macOS und Windows können Uno-Nutzer prinzipiell auch auf die Mobilbetriebssysteme Android und iOS zurückgreifen; außerdem gibt es auch ein WebAssembly-Frontend, das es .NET-Entwicklern ermöglicht, Webapplikationen mit JavaScript stressfrei zu erstellen.

Dieser Artikel wirft einen Blick auf die Möglichkeiten, die das Uno-Framework zur Verfügung stellt, und darauf, wie sich die ersten Schritte in die Welt von Uno gestalten und anfühlen.

Eine Frage der Arbeitsumgebung

Moderne plattformübergreifende .NET-Systeme beschränken den Entwickler im Allgemeinen nicht mehr auf die Arbeit unter Windows. Im Fall von Uno präsentiert sich die Abdeckung der unter Windows nutzbaren IDEs so, wie es Bild 1 von der Uno-Website zeigt [1].

Schon im Interesse der Bequemlichkeit empfiehlt sich für die folgenden Ausführungen Visual Studio Community 2022; die genaue Version der vom Autor auf der Windows-10-Workstation installierten IDE lautet 17.4.0 Preview 1.0.

Im Allgemeinen sind Windows und Linux ebenbürtig. Der bequemste Weg für die Entwicklung für macOS führt logischerweise zu Visual Studio für Mac. Im Bereich der IDE-Komponenten des Visual Studio Installer schlägt das Entwicklerteam folgende vor:

- ASP.NET und Webentwicklung,
- mobile Entwicklung mit .NET (Xamarin),
- Entwicklung für die universelle Windows-Plattform.

Bei der Arbeit mit Visual Studio 2022 wählen Sie die Option *NET Multi-Platform App UI-Entwicklung*. Beim Schreiben dieses Artikels war es ärgerlich, dass Uno noch auf .NET in der Version 5.0 setzt – dazu ist dessen SDK zu installieren, obwohl es von Microsoft nicht mehr mit Sicherheits-Updates ausgestattet wird [2]. Der Installationsassistent für Visual Studio wird sich übrigens in Zukunft bei Updates der IDE darüber beklagen, die Dialoge lassen sich allerdings (zumindest derzeit) noch abnicken.

Da beim plattformübergreifenden Entwickeln Konfigurationsprobleme mehr oder weniger zur Tagesordnung gehören, stellt das Entwicklerteam mit Uno-Check ein Überprüfungswerkzeug zur Verfügung; es lässt sich in der Eingabeaufforderung folgendermaßen bereitstellen:

```
C:\Users\tamha>dotnet tool install -g uno.check
```

Aufgerufen wird das Tool dann mit dem Befehl *uno-check*. Nach dem Bestätigen der UAC-Anforderung startet uno-check wie in Bild 2 gezeigt in einem neuen Fenster, das sogleich unter anderem über Probleme der Systemkonfiguration informiert. Auf der Workstation des Autors war beispielsweise das Android-SDK veraltet, was das Werkzeug allerdings direkt zu beheben versucht.

Allerdings sind nicht alle von dem Tool gefundenen Fehler gleichermaßen kritisch. Wer sich beispielsweise nicht für Linux interessiert, kann die Warnung über das Fehlen des Windows-Subsystem für Linux bedenkenlos ignorieren.

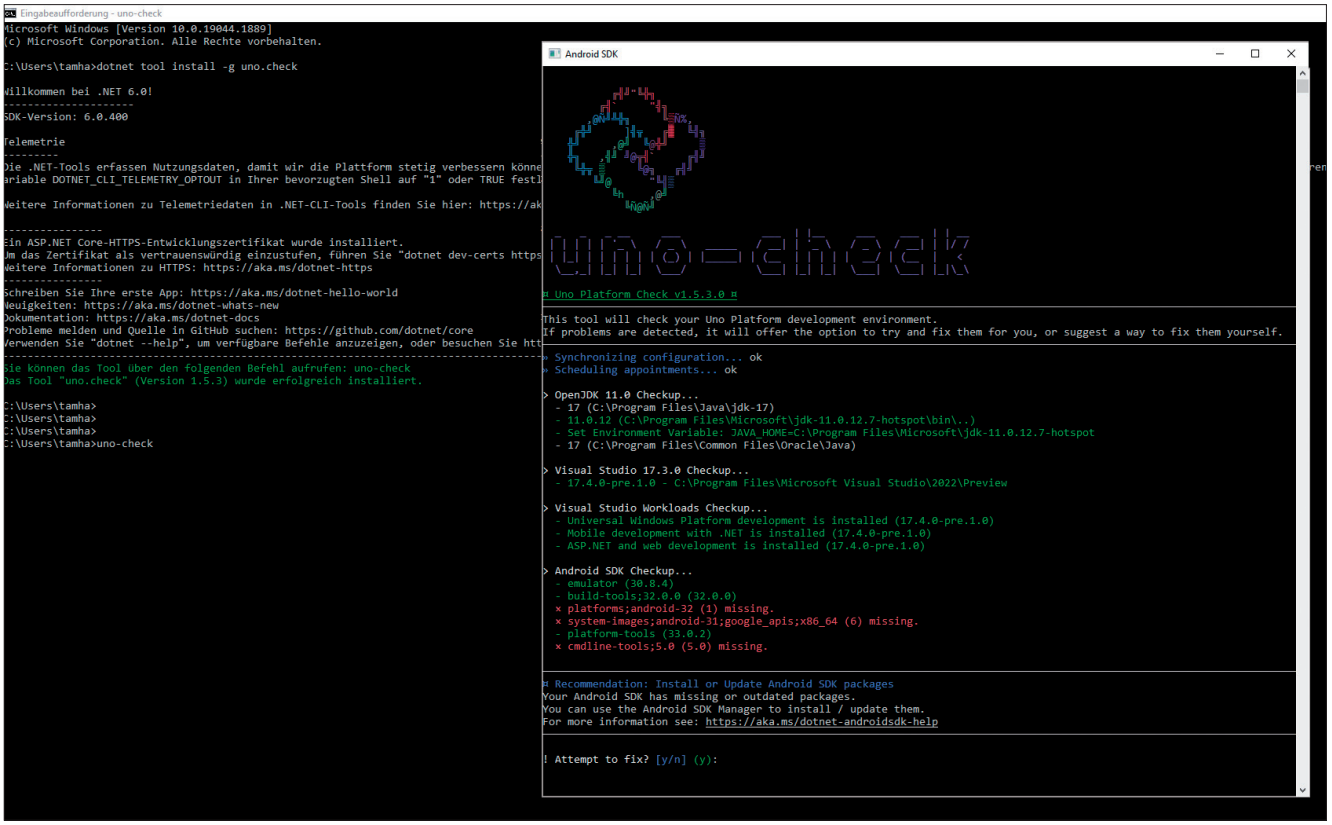
Der letzte Akt ist dann das Öffnen des Menüpunkts *Erweiterungen | Erweiterungen verwalten* in Visual Studio, wo Sie nach der Zeichenfolge „uno platform“ suchen. Lassen Sie danach das von Uno bereitgestellte Template-Paket (*Uno Platform Solutions Templates*) herunter und führen Sie den üblichen Neustart der IDE aus, um die neuen Projektvorlagen und Werkzeu-

Target platform coverage by IDE on Windows

	Windows 10/11 (UWP/WinUI)	Android	iOS	Web (WebAssembly)	macOS (Xamarin)	macOS (Skia-Gtk)	Linux (Skia-Gtk)	Windows 7+ (Skia-WPF)
Visual Studio	✔	✔	✔†	✔	✘	✔	✔	✔
VS Code	✘	✘	✘	✔	✘	✔	✔	✔
Codespaces / Gitpod	✘	✘	✘	✔	✘	✔	✔	✔
JetBrains Rider	✔	✔	✔†	✔	✘	✔	✔	✔

† You'll need to be connected to a Mac to run and debug iOS apps from Windows.

Visual Studio bietet die umfangreichste Systemabdeckung für Uno-Entwickler (Bild 1)



uno-check startet in einem eigenen Fenster und prüft die Systemkonfiguration (Bild 2)

ge in den Kompilationsprozess der Entwicklungsumgebung einzubinden.

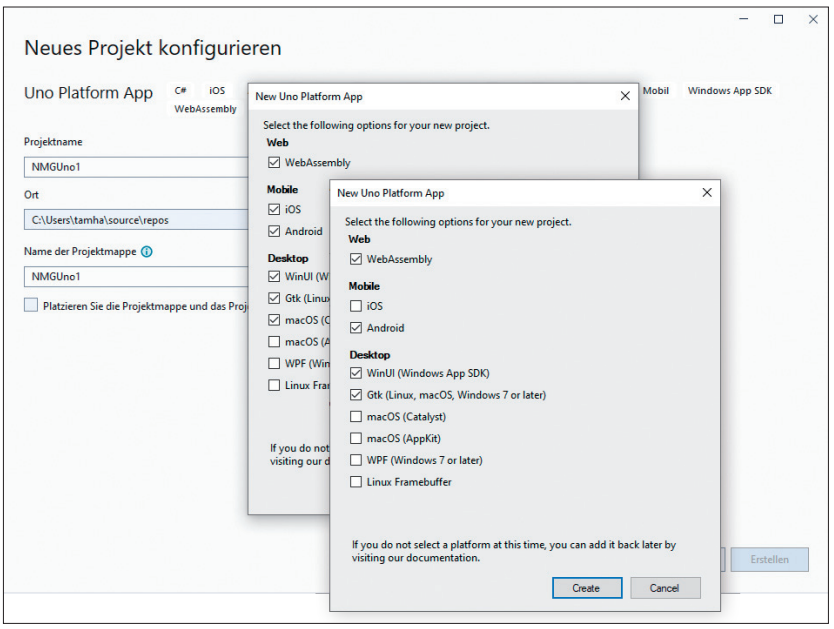
Inbetriebnahme der Arbeitsumgebung

Im nächsten Schritt wird Visual Studio wieder gestartet, um die Option zum Erzeugen eines neuen Projekts aufzurufen. Die Suche nach der Zeichenfolge „uno“ bringt dann drei Optionen zum Vorschein. Neben der zum Erstellen von Bibliotheken vorgesehenen Uno Plattform Library gibt es auch zwei Varianten einer Uno-Plattform-App. Die beiden unterscheiden sich vor allem im Bereich der Zielplattform unter Windows. Während die Vorlage *Uno Platform App* das Windows App SDK, WebAssembly, Android, iOS, Mac Catalyst, macOS und Linux unterstützt, sind es bei der Vorlage *Uno Platform App (Xamarin, UWP)* WebAssembly, Android, iOS, macOS, Linux und UWP.

Das Beispiel des Autors ist eine gewöhnliche Anwendung für Uno mit Namen *NMGUNO1*. Nach dem Anklicken des *Erstellen*-Buttons erscheint der in Bild 3 im Hintergrund gezeigte Dialog zur Auswahl der gewünschten Zielplattformen. Das Einpflegen von neuen Zielplattformen, nachdem das Projekts erzeugt ist, gestaltet sich etwas aufwendiger [3], weshalb es sich empfiehlt, bei der Zielauswahl nicht zu geizen.

Aus Demonstrationsgründen hat sich der Autor in den folgenden Schritten allerdings – wie in Bild 3 im Vordergrund gezeigt – nur für eine vergleichsweise eingeschränkte Auswahl entscheiden.

Nach dem Anklicken des *Create*-Buttons beginnt die IDE mit der Arbeit. Nach dem vollständigen Start bestätigen ▶



Der unter Windows 10 maximal mögliche Reichweitengrad (hinten) und die etwas pragmatischere Auswahl des Autors (vorn) (Bild 3)

Sie bei einem frisch installiertem Visual Studio unter anderem die Android-Lizenzen. Angesichts der Komplexität des Projekts ist es auf jeden Fall empfehlenswert, einige Zeit zu warten und eventuell sogar noch eine erneute Kompilierung zu starten.

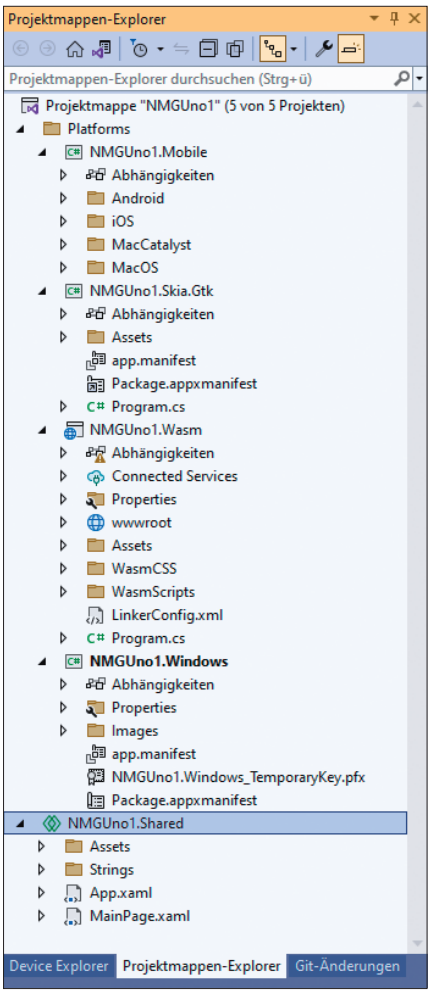
Sonstige fehlende Komponenten moniert die IDE dann im Projektmappen-Explorer, der das direkte Aktivieren des Visual Studio Installers erlaubt. Um Uno effizient zu nutzen, sollten Sie mindestens 30 GB Festspeicher einplanen.

Nach dem erfolgreichen Start der IDE präsentiert sich ein Projektskelett, wie es in Bild 4 gezeigt ist.

Das Projekt *NMGUno1.Windows* ist dabei die Windows-spezifische Variante, welche die für das Ausführen des Projekts unter Windows notwendige Intelligenz beisteuert. Hier befindet sich unter anderem die Manifestdatei, die das Kompilat gegenüber dem Betriebssystem ausweisen wird.

In *NMGUno1.Shared* findet sich dagegen der allgemeine Teil, der von allen Plattformen geteilt wird – denken Sie an die eigentliche Applikationslogik und/oder die Formulare, die dem Nutzer anzuzeigen sind.

Zur Illustrierung wird an dieser Stelle die Projektmappe *NMGUNO1.Windows* kompiliert. Lohn der Mühen ist die Anzeige des in Bild 5 gezeigten Hello-World-Fensters.



Uno-Projektskelette weisen einen beeindruckenden Codeumfang auf (Bild 4)

Den Code im Blick

Als Nächstes bietet es sich an, die einzelnen Elemente des Uno-Projekts Schritt für Schritt zu betrachten. Ein Blick auf Bild 4 genügt, um sich davon zu überzeugen, dass der Windows-spezifische Teil der Projektmappe keinen Einsprungpunkt bietet.

Schon hier sei angemerkt, dass dies nicht für alle Varianten von Uno gilt. Der für den Webbrowser vorgesehene WebAssembly-Teil (das Projekt *NMGUno1.Wasm*) hat mit der Datei *Program.cs* beispielsweise einen eigenen Einsprungpunkt, der nach folgendem Schema Renderer und Wrapper für XAML-Applikationen anwirft:

```
public class Program {
  private static App _app;
  static int Main(string[] args) {
    Microsoft.UI.Xaml.Application.
      Start(_ => _app = new App());
    return 0;
  }
}
```

Allen Applikationen gemein ist an dieser Stelle wiederum, dass der Code der Datei *App.xaml.cs* Code im Rahmen des Programmstarts ausgeführt wird. Neben dem Einrichten verschiedener Eventhandler für das Entgegennehmen von Betriebssystemereignissen ist auch die Methode *InitializeLogging()* interessant, die verschiedene Arten des Protokollierens aktiviert und/oder deaktiviert (siehe auch den Kasten Logging ist langsam).

Der eigentliche Aufbau des Logging-Systems ist auch im Allgemeinen interessant. Analog zum von Google in Android eingebauten LogCat gilt auch im Fall von Uno, dass der Entwickler mit verschiedenen Bedingungen festlegen darf, welche Informationen in den Log-Dateien niederzulegen sind.

Das eigentliche Erzeugen der Log-Dateien erfolgt dabei durch einen Provider, der nach folgendem Muster in eine *LoggerFactory*-Klasse einzuschreiben ist; dabei ist jeder Provider für eine Plattform oder ein Zielsystem verantwortlich:

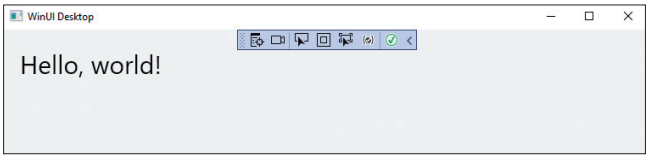
```
private static void InitializeLogging(){
  #if DEBUG
    var factory = LoggerFactory.Create(builder =>
    {
      builder.AddProvider(new global::
        Uno.Extensions.Logging.WebAssembly.
        WebAssemblyConsoleLoggerProvider());
```

Im nächsten Schritt erfolgt dann die Einschränkung durch verschiedene Filter:

```
builder.SetMinimumLevel(LogLevel.Information);
builder.AddFilter("Uno", LogLevel.Warning);
builder.AddFilter("Windows", LogLevel.Warning);
```

Logging ist langsam

Die Deaktivierung der Logging-Funktionen des Uno-Frameworks ist keine Boshaftigkeit des Entwicklerteams. Auf manchen Plattformen verlangsamt die Verarbeitung der aufgezeichneten Daten das System deutlich [9].



Das Beispielprojekt funktioniert unter Windows problemlos (Bild 5)

Neben *SetMinimumLevel()*, das auf alle angemeldeten Provider gleichermaßen angewendet wird, können mit der Methode *AddFilter()* auch Provider-spezifische Richtlinien angelegt werden. Würden Sie beispielsweise nach einem Fehler in dem Windows-Teil der Applikation suchen, so könnte es empfehlenswert sein, dort einen „kommunikativeren" Log-Level festzulegen.

Wie dem auch sei: Im nächsten Schritt können Sie sich der Startdatei zuwenden. *MainPage.xaml* enthält dabei mehr oder weniger gewöhnliches XAML-Markup, das die Nutzerschnittstelle umsetzt:

```
<Page
  x:Class="NMGUno1.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
         presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  ...
  Background="{ThemeResource
              ApplicationPageBackgroundThemeBrush}">
  <Grid>
    <TextBlock Text="Hello, world!" Margin="20"
      FontSize="30" />
  </Grid>
</Page>
```

Schon an dieser Stelle sei angemerkt, dass der Wysiwyg-Editor der Entwicklungsumgebung bei der Bearbeitung von für Uno vorgesehenen XAML-Dateien deaktiviert ist; die UIs entstehen also von Hand durch manuelles Niederschreiben von Markup.

Im Hintergrund orientiert sich das Uno-Entwicklerteam dabei im Allgemeinen wie in Bild 6 [4] gezeigt an WinUI.

Auf den meisten Plattformen verwendet Uno dabei die .NET-Funktionen direkt. Die einzige Ausnahme sind unixoide Betriebssysteme, bei denen die Skia-Rendering-Engine für die Anzeige eingespannt wird.

Interessant ist auch noch der Code-behind, in dem sich wie folgt die Inklusion verschiedener für .NET-Entwickler bekannter Namensräume aus dem Bereich der GUI-Entwicklung wiederfindet:

```
using Microsoft.UI.Xaml;
using Microsoft.UI.Xaml.Controls;
using Microsoft.UI.Xaml.Controls.Primitives;
```

Die eigentliche Initialisierung der *MainPage* erfolgt dann ebenfalls so, wie es von verschiedenen anderen XAML-basierten Plattformen zu erwarten wäre:

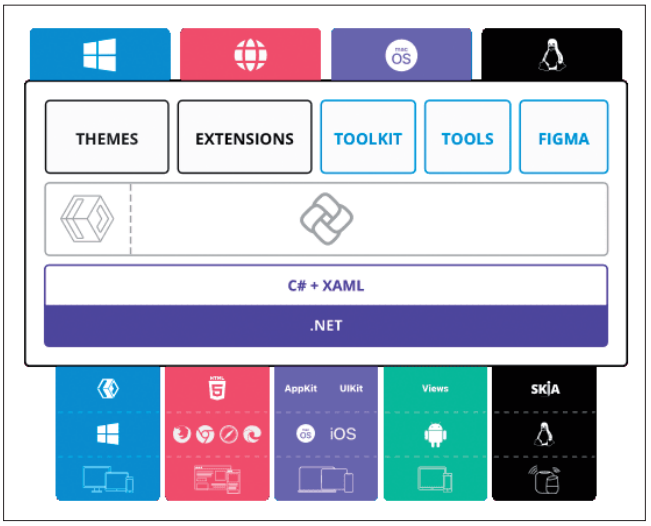
```
public sealed partial class MainPage : Page
{
  public MainPage()
  {
    this.InitializeComponent();
  }
}
```

Als kleine Maßnahme zur Familiarisierung platziert der nächste Schritt eine Schaltfläche im Formular, die beim Anklicken Ereignisse auslöst.

Bei der Arbeit mit Uno ist es immer empfehlenswert oder Konvention, zur Verbindung zwischen dem mit XAML Dargestellten und dem Code-behind auf die Datenbindung zu setzen. Deshalb ist folgende Anpassung notwendig:

```
<Grid>
  <TextBlock x:Name="TamsText" Text="Hello, world!"
    Margin="20" FontSize="30" />
  <Button Content="NMG-Testknopf"
    Click="{x:Bind OnClick}" />
</Grid>
```

Zum einen erhält das vom Projektgenerator erzeugte <*TextBlock*>-Element einen Namen, um es für Code ansprechbar zu machen. Zum anderen erweitern Sie das Grid-Steuerelement um eine Schaltfläche, die neben der *Content*-Eigenschaft auch ein *Click*-Attribut erhält. In ihm findet sich die – in geschweifte Klammern zu setzende – Bindungszeichenfolge, die dem Schema *{x:Bind OnClick}* entspricht.

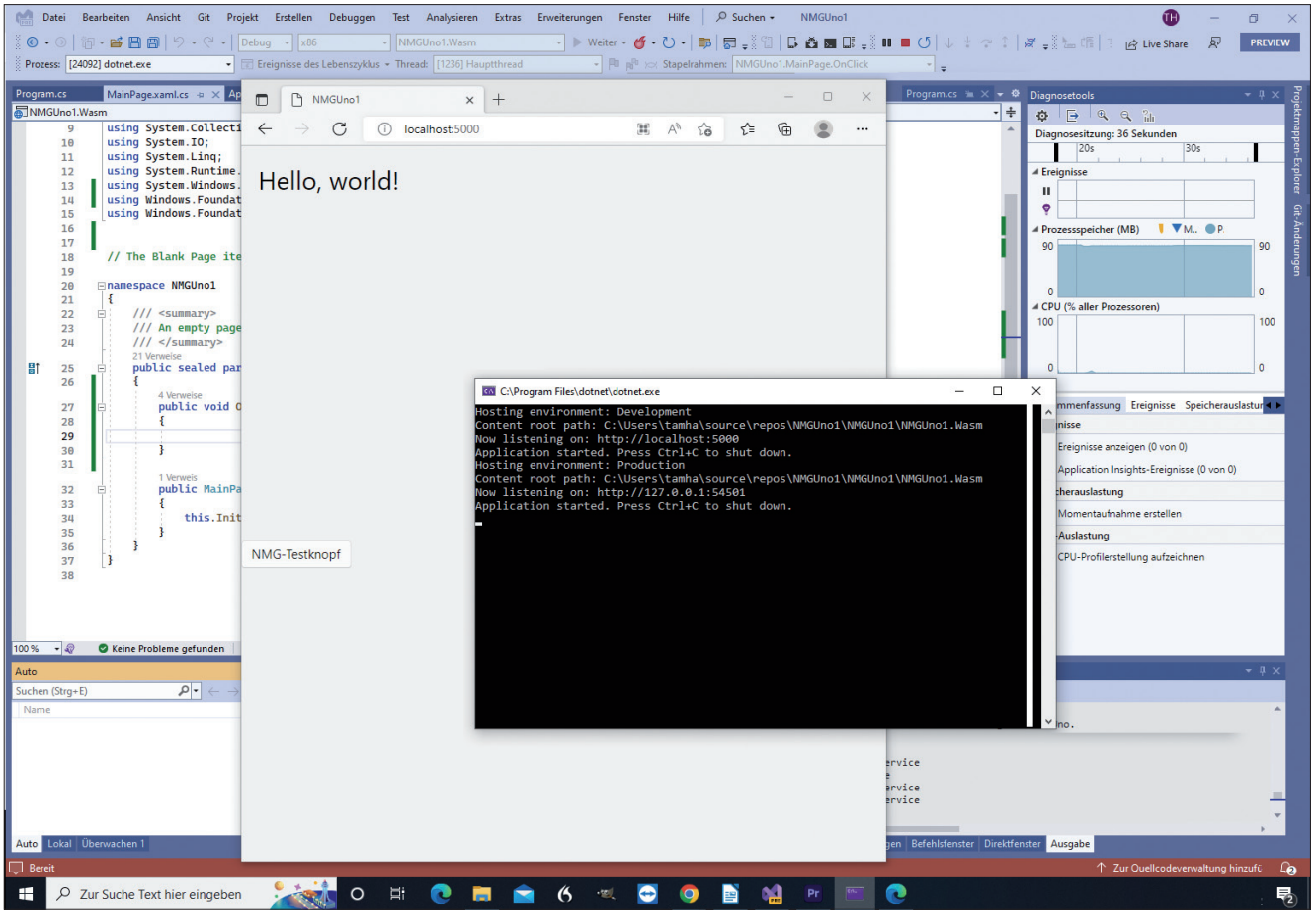


WinUI steht im Mittelpunkt der Arbeit mit Vue (Bild 6)

Im nächsten Schritt können Sie auch schon in den Code-behind wechseln und einen grundlegenden Eventhandler anlegen:

```
public sealed partial class MainPage : Page {
  public void OnClick() {
    TamsText.Text = "Hello NMG!";
  }
```

Beachten Sie hierbei bitte, dass die Verbindung zwischen dem Visual-Studio-Compiler und den Uno-spezifischen Teilen im Bezug auf Datenbindungsfehler alles andere als perfekt ist. Ein ungültiger Binding-String führte im Test des Autors immer wieder zu seltsamstem Fehlverhalten, bei dem ▶



Die .NET-Applikation läuft dank Uno-Framework auch im Browser problemlos (Bild 7)

der Compiler beispielsweise an sich bekannte Namensräume monierte.

Trotz allem ist an dieser Stelle eine Kompilierung erforderlich. Klicken Sie danach im kompilierten Beispielprogramm die Schaltfläche an, um sich – ganz wie unter vertrauten Windows-Plattformen – an einer Anpassung der angezeigten Nachricht zu erfreuen.

Isolierung von plattformspezifischem Code

In einer idealen Welt würde ein plattformübergreifendes Framework alle Funktionen umfassen, die der Entwickler für seine Anwendung benötigt. In der Praxis ist dies schon aufgrund der extremen Diversität der Funktionen nicht möglich; außerdem gilt seit den Zeiten von Palm (Stichwort „Zen of Palm“ [5]), dass es nicht vernünftig ist, ein Nutzerinterface oder eine Logik für verschiedene Arten von Schnittstellen zur Verfügung zu stellen.

Im nächsten Schritt soll im Plattform-Auswahldialog von Visual Studio die Option *NMGUno1.Wasm* gewählt werden, um den WebAssembly-Kompilationspfad des Uno-Frameworks zu aktivieren. Sodann führt ein Klick auf das *Run*-Symbol zu dem in Bild 7 gezeigten Verhalten.

Bei der Arbeit unter Windows 10 wird Visual Studio 2022 dabei Microsofts Edge-Browser starten. Beachten Sie außerdem, dass das Ingangsetzen der WebAssembly-Toolchain ein rechenleistungsintensiver Prozess ist, der auf der Workstation des Autors etwa doppelt so viel Zeit in Anspruch nimmt wie das Laden einer „gewöhnlich“ kompilierten UWP-App. Bei der Arbeit mit plattformübergreifenden Frameworks ist es im Allgemeinen immer zu empfehlen, den Hauptteil der Entwicklung auf der am bequemsten zu handhabenden Plattform auszuführen und andere Plattformen nur im Rahmen der Qualitätssicherheit zu aktivieren.

Als nächster Schritt steht die Auseinandersetzung mit Möglichkeiten zum Targeting bestimmter Plattformen an. Der einfachste Weg besteht darin, den Präprozessor zu nutzen, der laut der Dokumentation die in Tabelle 1 gezeigten Konstanten anbietet.

Im nächsten Schritt wollen wir das Uno-Entwicklerteam beim Wort nehmen und die in der Windows- und in der WebAssembly-Version in dem *TextBox*-Element eingetragenen Texte anpassen. Dies lässt sich durch eine nach dem folgenden Schema angepasste *OnClick()*-Variante bewerkstelligen:

```
public void OnClick() {
#if NETFX_CORE
  TamsText.Text ="Hallo Windows";
#elif HAS_UNO_WASM
  TamsText.Text = "Hallo WASM";
#endif
}
```

Ein Testlauf führt an dieser Stelle leider dazu, dass nur noch die WebAssembly-Version den im *TextBlock* angezeigten String verändert – unter Windows verpuffen die Klickereignisse ersatzlos.

Die Erklärung für dieses auf den ersten Blick seltsam erscheinende Verhalten findet sich in der zu Uno gehörenden Gitter-Community (siehe auch den Hinweis im Kasten Kommunikation per Chat). Der Nutzer @davidjohnoliver erklärte auf eine ähnlich geartete Nutzer-Anfrage:

„NETFX_Core is WinRT-specific, it's normally not defined for .NET Core projects."

An dieser Stelle bewahrheitet sich eine weitere alte Weisheit der plattformübergreifenden Entwicklung: Wer seine Host-Plattform gut kennt, kann sauberer programmieren. Eine funktionierende Variante der Windows-Erkennung lässt sich folgendermaßen ausprogrammieren:

```
public void OnClick() {
  #if WINDOWS_UWP || NETFX_CORE ||  WINDOWS
    TamsText.Text ="Hallo Windows";
  #elif HAS_UNO_WASM
    TamsText.Text = "Hallo WASM";
  #endif
}
```

Beachten Sie, dass die Dokumentation immer wieder die Verwendung des Präprozessor-Flags *HAS_UNO* empfiehlt. In Tests des Autors führt es allerdings schon deshalb nicht zum Ziel, weil es alle Uno-Systeme gleichermaßen anzieht, also auch unter WebAssembly *True* zurückliefern würde.

Da das übermäßige Verwenden von Präprozessor-Definitionen nicht nur in der C-Welt zu den Antimustern gehört, empfiehlt es sich, ab einem gewissen Komplexitätsgrad auf die verschiedenen im .NET-Standard integrierten Hilfswerkzeug zu setzen. Ein Klassiker sind dabei die *using*-Aliase. Im Prinzip handelt es sich dabei um eine nach dem folgenden Schema aufgebaute Kaskade aus *using*-Statements, die unter Verwendung des Präprozessors je nach Zielplattform einen anderen Namensraum laden:

```
#if __ANDROID__
using _View = Android.Views.View;
#elif __IOS__
using _View = UIKit.UIView;
#else
using _View = Windows.UI.Xaml.UIElement;
#endif
```

Der eigentliche Zugriff auf die über *using* geladenen Elemente erfolgt dann nach folgendem für .NET-Entwickler im Allgemeinen bekannten Schema:

```
public IEnumerable<_View>
  FindDescendants(FrameworkElement parent) => ...
```

Interessant ist in diesem Zusammenhang auch die Nutzung partieller Klassen, die ihre Eigenschaften und/oder Methoden über mehrere Dateien verteilen können. Der C#-Compiler sucht sie sich dann im Rahmen des Zusammenbaus der eigentlichen Assembly aus mehreren Quellen zusammen. Auch dabei handelt es sich um ein Standardmerkmal der .NET-Sprachspezifikation. Microsoft bietet dazu ein detailliertes Tutorial an, das die Technologie samt Besonderheiten en détail durcharbeitet [6].

Kommunikation per Chat

Wer schnelle Hilfe bei einem das Uno-Framework betreffenden Problem sucht, ist in der Gitter-Community gut aufgehoben. Gitter ist – unwissenschaftlich gesagt – ein Slack-Analogon [10].

Konditionaler Austausch ganzer UI-Elemente

Die obige Anspielung auf Zen/Palm ist insbesondere im Handcomputerbereich relevant: Auch wenn die Rechenleistung moderner Android-Smartphones mit älteren Workstations mehr als mithalten kann, hinkt die allgemeine Haptik noch immer hinterher. Ein am Desktop mit Maus und Tastatur problemlos bedienbares Formular ist, insbesondere in Zeiten kapazitiver Bildschirme ohne Stift, für einen durchschnittlichen Anwender auf einem Smartphone nicht wirklich zu verwenden.

Das Uno-Framework macht sich an dieser Stelle die Flexibilität des XAML-Sprachstandards zunutze, um Entwicklern das Einbetten konditionaler Funktionalität direkt in den Markup-Code zu ermöglichen. Dreh- und Angelpunkt ist dabei ein vergleichsweise kompliziertes System aus Präfixen, das in Tabelle 2 übersichtlich dargestellt wird.

Ein Präfix sorgt dafür, dass das jeweilige Element auf allen Plattformen angewendet wird, die in der Tabelle in der Rubrik *Adressierte Plattformen* vorkommen. Kommt die XAML-Datei stattdessen auf einem der in der Spalte *Nicht berücksichtigte Plattformen* genannten Systeme zum Einsatz, so verfällt das jeweilige Markup-Stück ersatzlos. Wird also beispielsweise das Präfix *android* verwendet, so würde das jeweilige Steuerelement nur unter Android sichtbar sein; auf allen anderen Plattformen käme es nicht zum Einsatz. ▶

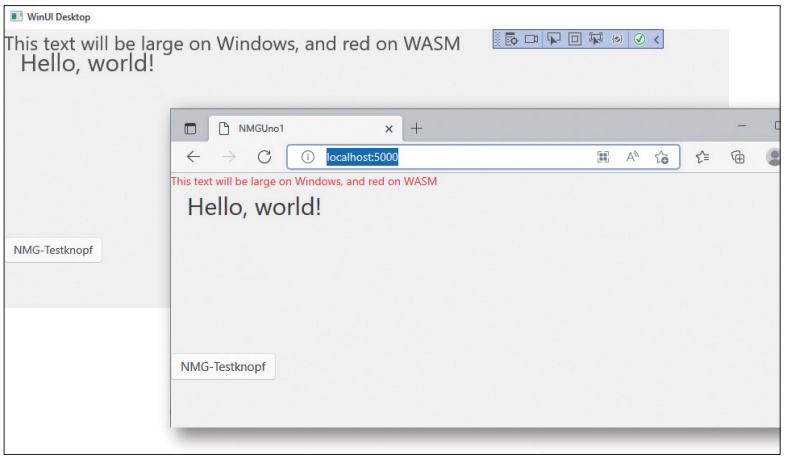
Tabelle 1: Konstanten des Präprozessors

Plattform	Symbol
UWP	*NETFX_CORE*
Android	*__ANDROID__*
iOS	*__IOS__*
WebAssembly	*HAS_UNO_WASM*
macOS	*__MACOS__*
Skia	*HAS_UNO_SKIA*

Zu jedem Präfix gehört außerdem noch ein Namensraum-String, siehe Tabelle. Wichtig ist außerdem auch noch das Feld *In mc:Ignorable enthalten*, weil diese Angabe darüber entscheidet, ob der jeweilige Namensraum in das *Ignore*-Feld der XAML-Dateien aufgenommen werden muss.

Im Prinzip geht es dabei darum, den Parser daran zu hindern, für ihn nicht vorgesehene Namensräume zu verarbeiten und dabei zu verenden. Aus diesem Grund müssen Sie alle Namensräume, die nicht für den Windows-Parser vorgesehen sind, in *mc:Ignorable* ablegen, sonst kommt es zu Compiler-Fehlern.

Ein vergleichsweise kompletter Header würde dann folgendermaßen aussehen:



Unter Windows ist das Feld „This Text ...“ relativ groß (hinten), während der Text unter WebAssembly in roter Farbe erscheint (vorne) (Bild 8)

```
<Page x:Class="HelloWorld.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
         presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:android="http://uno.ui/android"
  xmlns:ios="http://uno.ui/ios"
  xmlns:wasm="http://uno.ui/wasm"
  xmlns:win="http://schemas.microsoft.com/winfx/2006/
             xaml/presentation"
  xmlns:not_android="http://schemas.microsoft.com/winfx/
                     2006/xaml/presentation"
  xmlns:d="http://schemas.microsoft.com/expression/
           blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/
            markup-compatibility/2006"
  mc:Ignorable="d android ios wasm">
```

Neben dem Einbinden der verschiedenen Präfixe mit ihren in der Tabelle jeweils genannten URLs zeigt sich hier auch, dass *android*, *ios* und *wasm* zwecks Befriedung des Windows-Parsers und seiner Syntaxüberprüfung im Attribut *mc:Ignorable* aufgenommen wurden.

Nach diesen ziemlich theoretischen Überlegungen soll es nun an die Praxis gehen. Dazu soll das auf WebAssembly und Windows basierende Projekt um eine Weiche im Bereich der Nutzeroberfläche ergänzt werden. Zuerst ist hierfür in der

Tabelle 2: Präfixe und Konfiguration der Zielplattform(en) mithilfe des Präprozessors

Präfix	Adressierte Plattformen	Nicht berücksichtigte Plattformen	Namensraum	In mc:Ignorable enthalten
win	Windows	Android, iOS, Web, macOS, Skia	http://schemas.microsoft.com/winfx/2006/xaml/presentation	*no*
xamarin	Android, iOS, Web, macOS, Skia	Windows	http://uno.ui/xamarin	*yes*
not_win	Android, iOS, Web, macOS, Skia	Windows	http://uno.ui/not_win	*yes*
android	Android	Windows, iOS, Web, macOS, Skia	http://uno.ui/android	*yes*
ios	iOS	Windows, Android, Web, macOS, Skia	http://uno.ui/ios	*yes*
wasm	Web	Windows, Android, iOS, macOS, Skia	http://uno.ui/wasm	*yes*
macos	macOS	Windows, Android, iOS, Web, Skia	http://uno.ui/macos	*yes*
skia	Skia	Windows, Android, iOS, Web, macOS	http://uno.ui/skia	*yes*
not_android	Windows, iOS, Web, macOS, Skia	Android	http://schemas.microsoft.com/winfx/2006/xaml/presentation	*no*
not_ios	Windows, Android, Web, macOS, Skia	iOS	http://schemas.microsoft.com/winfx/2006/xaml/presentation	*no*
not_wasm	Windows, Android, iOS, macOS, Skia	Web	http://schemas.microsoft.com/winfx/2006/xaml/presentation	*no*
not_macos	Windows, Android, iOS, Web, Skia	macOS	http://schemas.microsoft.com/winfx/2006/xaml/presentation	*no*
not_skia	Windows, Android, iOS, Web, macOS	Skia	http://schemas.microsoft.com/winfx/2006/xaml/presentation	*no*

Plattformspezifisches Targeting

Mit Android arbeitende Entwickler vermissen mitunter die Möglichkeit, durch Dateinamenerweiterungen bestimmte Ressourcen mit bestimmten Plattformen zu verbinden. Uno realisiert ein ähnliches System, welches das Uno-Team detailliert beschreibt [11].

XAML-Datei nach dem folgenden Schema der Header anzupassen:

```
<Page
  ...
  mc:Ignorable="d wasm"
  xmlns:wasm="http://uno.ui/wasm"
  xmlns:win="http://schemas.microsoft.com/winfx/2006/
             xaml/presentation">
```

Ein Blick auf die Tabelle informiert darüber, dass der WASM-Namensraum als Teil der „Ignorables“ aufgelistet werden muss.

Visual Studio erschwert die Arbeit an dieser Stelle insofern, als die Datei bereits ein *Ignorable*-Attribut enthält, das nach dem hier gezeigten Schema zu erweitern ist.

Im nächsten Schritt lässt sich dann ein *TextBlock* erzeugen, der die Attribute *win:FontSize* und *wasm:Foreground* konditional lädt:

```
<TextBlock Text="This text will be large on Windows,
                 and red on WASM"
  win:FontSize="24"
  wasm:Foreground="Red"
  TextWrapping="Wrap"/>
```

Wenn Sie das Programm danach sowohl unter Windows als auch unter WebAssembly ausführen, sehen Sie das in Bild 8 gezeigte Ergebnis.

Angemerkt sei, dass das konditionale Laden nicht auf das Beeinflussen von Attributen beschränkt ist. Es ist auch möglich, ein ganzes Steuerelement zu deklarieren, das nur für eine bestimmte Plattform gedacht ist:

```
<wasm:TextBlock Text="Erscheint
        nur unter WebAssembly" />
```

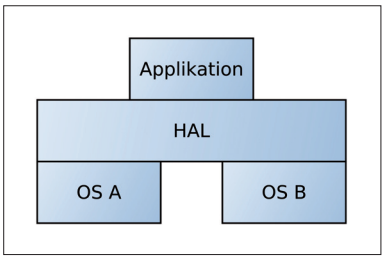
Fortgeschrittene APIs

Plattformübergreifende Frameworks gelten nur allzu oft als Werkzeuge, um Geschäftsapplikationen umzusetzen, die keine tiefere Bindung zu Hardwarefunktionen eingehen können, wie beispielsweise ein Beschleunigungssensor (repräsentiert durch die Klasse *Accelerometer*).

Uno löst dieses Problem insofern, als das Produkt – teilweise auch im Zusammenspiel mit Xamarin – Abstraktionsschichten anbietet, welche die Interaktion mit nativen Hardwareelementen ohne plattformspezifischen Code ermöglichen (Bild 9).

Für Zugriff auf den Beschleunigungssensor und Co. empfiehlt sich dabei der Besuch der Uno-Webseite „Accelerometer“ [7]. Dort befindet sich in der rechts eingeblendeten Navigation unter der Rubrik *Non-UI APIs* eine Liste der verschiedenen Funktionsgruppen, bei denen das Uno-Framework plattformunabhängige Zugriffsklassen zur Verfügung stellt.

Besonders interessant ist in diesem Zusammenhang die – wahrscheinlich von Qt entlehnte – Möglichkeit, Ressourcen direkt im geteilten Projekt unterzubringen. Diese Funktion unterscheidet sich insofern von dem von Xamarin und Co. bekannten Merkmal, als eine Ressource für alle Varianten des Projekts angeboten wird. Andererseits eignet sich die Funktion nicht, um Splashscreens, Icons und andere direkt vom jeweiligen Host-Betriebssystem anzusprechende Ressourcen zur Verfügung zu stellen.



Der Hardware Abstraction Layer (HAL) erspart dem Entwickler Arbeit (Bild 9)

Ärgerlich ist dabei, dass es im Bereich der unterstützten Bildformate zwischen den verschiedenen Plattformen teilweise erhebliche Unterschiede gibt (Bild 10). Der Kasten Plattformspezifisches Targeting weist auf eine andere Besonderheit hinsichtlich verschiedener Plattformen hin.

Zur Demonstration der Möglichkeiten verwendet das folgende Beispiel das JPEG-Bild *IMG_20220912_191812_0.jpg*. Geteilte Ressourcen werden dabei im *NMGUno1.Shared*-Projekt abgelegt, siehe Bild 4. Öffnen Sie es im Projektmappen-Explorer und klicken Sie den *Assets*-Ordner mit der rechten Maustaste an, um die Datei danach per Drag-and- ▶

Supported asset types

At the moment, only the following image file types are supported:

	.bmp (Win BMP)	.gif‡	.heic (Apple)	.jpg & .jpeg (JFIF)	.png	.webp	.pdf	.svg
Windows UWP	✔	✔	✘	✔	✔	✔	✘	✔
Android 10	✔	✔‡	✔	✔	✔	✔	✔	✔
iOS 13	✔	✔‡	✔	✔	✔	✘	✘	✘
macOS	✔	✔‡	✔	✔	✔	✘	✔	✘
Wasm†	✔	✔‡	✘†	✔	✔	✘†	✘†	✔
Skia WPF	✔	✔‡	✘	✔	✔	✔	✘	✘

- † Actual **Wasm image format support** is browser dependent. For example, `.webp` is not working on Safari on macOS, but works on Chromium-based browsers. Checkmarks (✔) indicates a format that can safely expected to work on all browsers able to run Wasm applications.
- ‡ **Gif animation support**:
  - Play/Pause not implemented in Uno yet
  - Always animated on Wasm
  - Not animated on other Uno platforms

Nicht jedes Betriebssystem unterstützt jedes Bildformat (Bild 10)

drop abzulegen. Wichtig ist, als *Buildvorgang* wie in Bild 11 gezeigt die Option *Inhalt* auszuwählen.

Für die Anzeige der Ressourcen reicht es dann aus, ein *Image*-Steuerelement in die XAML-Datei einzupflegen:

```
<Grid>
  ...
  <Image x:Name="myXAMLImageElement"
    Source="ms-appx:///Assets/
          IMG_20220912_191812_0.jpg"/>
```

Interessant ist hier vor allem das Präfix *ms-appx:///*, das UWP-Entwicklern von der Arbeit mit dem UWP-Ressourcensystem her bekannt vorkommen dürfte.

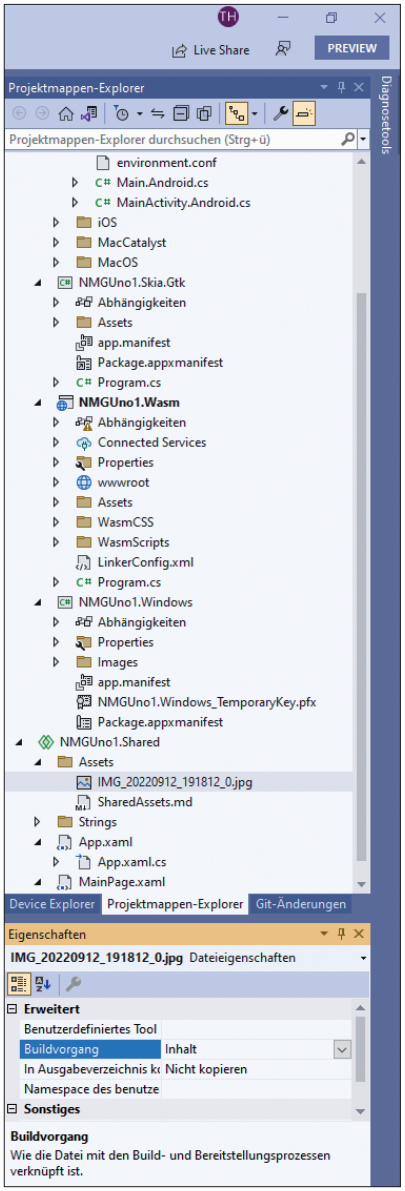
WebAssembly-Spezifikum: Interaktion mit Cookies

Als letzte Aufgabe soll dieser Artikel noch Möglichkeiten betrachten, um die WebAssembly-Variante der Applikation mit zusätzlichen browserspezifischen Funktionen aufzurüsten.

Am wichtigsten dürfte dabei die Möglichkeit zum Anlegen von Cookies sein – analog zu einer gewöhnlichen Webseite ist es auch unter Uno möglich, Informationen im Client zu persistieren.

Da die Programmierschnittstelle der WebAssembly-Runtime im Prinzip auf Webtechnologien setzt, dürfte der Funktionsumfang des folgenden Codebeispiels JavaScript-erfahrenen Entwicklern bekannt vorkommen:

```
#if __WASM__
var cookie = new Cookie("MyCookie", "somevalue");
var request = new SetCookieRequest(cookie)
{
  Path = "/",
  Expires =
    DateTimeOffset.UtcNow.AddDays(2),
  Secure = true,
};
CookieManager.GetDefault().
SetCookie(request);
#endif
```



Das JPEG-Bild ist zur Auslieferung bereit (Bild 11)

Analog zur Arbeit mit gewöhnlichem Webcode gilt auch im Fall von Uno, dass das Anlegen eines Cookies im Browser einige Attribute voraussetzt. Diese werden normalerweise in Form einer *SetCookieRequest*-Klasse bereitgestellt, die durch Aufruf von *CookieManager.GetDefault().SetCookie()* in den Cookie-Speicher des Browsers der Workstation übernommen werden.

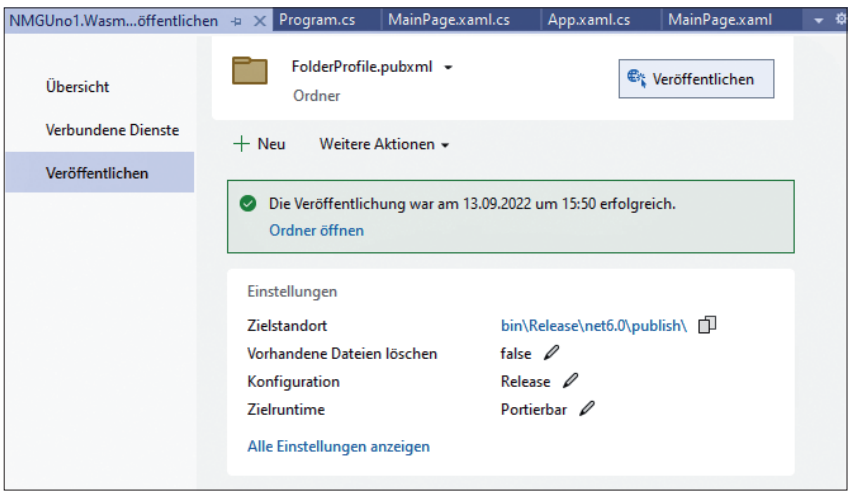
Wichtig ist in diesem Zusammenhang vor allem das Attribut *Expires*. Es legt fest, wann der Browser das Cookie selbsttätig löschen darf, um Speicherplatz auf dem Rechner oder dem Telefon freizugeben.

Das Lesen der vorhandenen Cookies erfolgt dann durch eine nach folgendem Schema aufgebaute Methode:

```
#if __WASM__
var cookies = CookieManager.
  GetDefault().GetCookies();
foreach (var cookie in cookies)
{
  Debug.WriteLine(cookie.Name);
  Debug.WriteLine(cookie.Value);
}
#endif
```

Auch das ist keine Raketenwissenschaft: *CookieManager.GetDefault().GetCookies()* liefert eine Liste aller für die Applikation zugänglichen Cookies. Die *foreach*-Schleife hat dann die Aufgabe, die Elemente nach außen zu tragen.

WebAssembly-Applikationen lassen sich außerdem – durchaus bequem – an Webbrowser ausliefern. Das Unter-Framework unterstützt dabei die Server Apache, IIS und Nginx gleichermaßen. Die wichtigste Aufgabe des Entwicklers ist dabei,



Visual Studio meldet: Die Paketierung der WebAssembly-Applikation verlief erfolgreich (Bild 12)

den Assistenten für das Veröffentlichen in Gang zu setzen, der sich im Projektmappen-Explorer nach einem Rechtsklick auf das Projekt *NMGUno1.Wasm* aktivieren lässt.

Wer wie der Autor testweise die Veröffentlichung in einen lokalen Ordner befiehlt, bekommt danach ein Verzeichnis wie *C:/Users/tamha/source/repos/NMGUno1/NMGUno1/NMGUno1.Wasm/bin/Release/net6.0/publish/* zurückgeliefert, das gut 40 MB an Daten enthält. Visual Studio wird außerdem wie in Bild 12 über das erfolgreiche Durchlaufen des Veröffentlichungsprozesses informieren.

Zur eigentlichen Auslieferung sind einige Operationen auszuführen, um den jeweiligen Webserver zum Verstehen der jeweiligen MIME-Typen und sonstigen Anforderungen der WebAssembly-Runtime zu befähigen. Auch hierzu hält die Uno-Website weitere Informationen zu den notwendigen Maßnahmen bereit [8].

Wer übrigens seine Applikation in der EU hostet oder in der EU eine zustellfähige Adresse hat, muss darauf achten, bei der Nutzung von Cookies die DSGVO zu beachten.

Fazit

Mit der Uno-Plattform haben .NET-Entwickler ein machtvolles Werkzeug in der Hand, um plattformübergreifende Applikationen zu erstellen. Dass dieses Produkt quasi komplett ohne die Unterstützung Microsofts entstand, zeigt, dass das Open-Source- und Drittanbieter-Ökosystem von .NET auch im Jahr 2023 effizient und lebendig ist. ■

[1] *Uno, Select your development environment, www.dotnetpro.de/SL2306Uno1*
[2] *.NET 5.0.17, www.dotnetpro.de/SL2306Uno2*
[3] *Uno, Adding Platforms to an Existing Project, www.dotnetpro.de/SL2306Uno3*
[4] *About the Uno Platform, www.dotnetpro.de/SL2306Uno4*
[5] *Zen of Palm, www.dotnetpro.de/SL2306Uno5*
[6] *Partielle Klassen und Methoden (C#-Programmierhandbuch), www.dotnetpro.de/SL2306Uno6*
[7] *Uno, Accelerometer, www.dotnetpro.de/SL2306Uno7*
[8] *Uno, Apache, www.dotnetpro.de/SL2306Uno8*
[9] *Uno.UI – Performance, www.dotnetpro.de/SL2306Uno9*
[10] *Gitter, www.dotnetpro.de/SL2306Uno10*
[11] *Assets and image display, www.dotnetpro.de/SL2306Uno11*

Tam Hanna
entwickelt Programme für verschiedene Plattformen, betreibt Online-Newsdienste zum Thema und steht für Fragen, Trainings und Vorträge gern zur Verfügung. Sie erreichen ihn unter der E-Mail-Adresse **tamhan@tamoggemon.com**.

dnpCode A2306Uno