PROGRAMMIEREN MIT WINUI 3

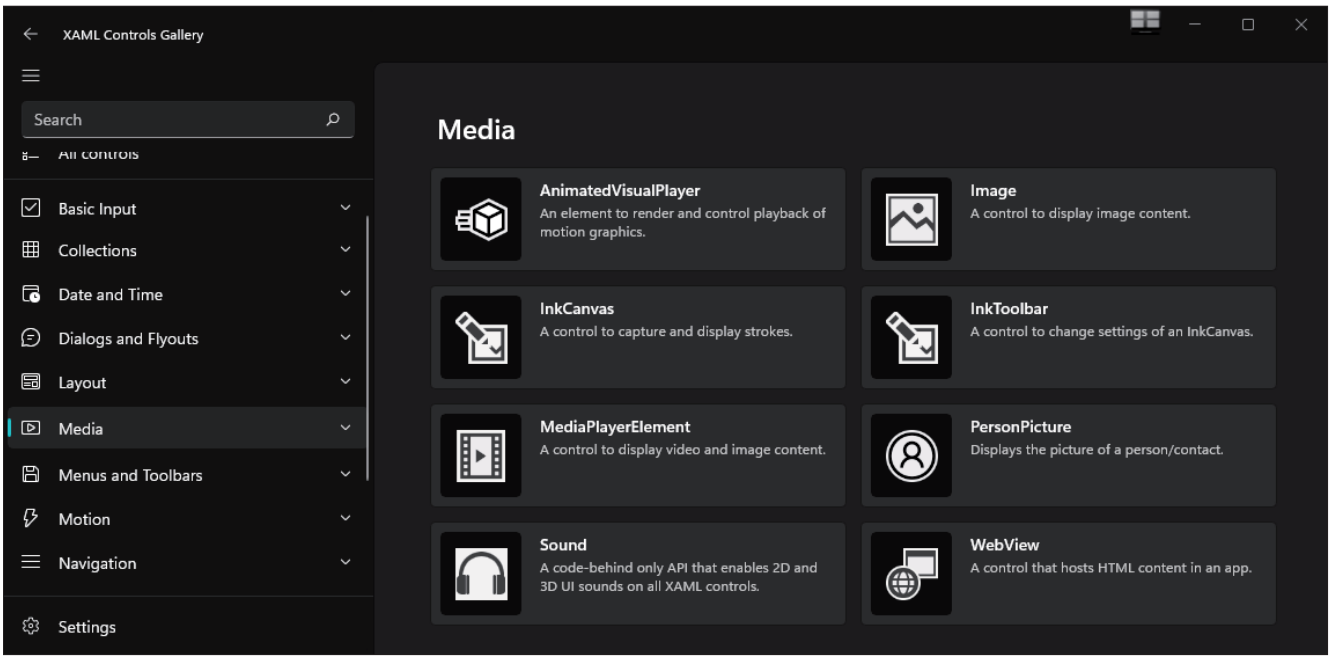
UI zu gewinnen

Auch wenn es so klingt: WinUI 3 ist kein Gewinnspiel. Doch die Microsoft-Technologie zur Programmierung von Windows-Desktop-Anwendungen kann für Sie zum Gewinn werden.

Um mit den Konkurrenten gleichziehen zu können, versah Microsoft damals Windows 8 mit einem App Store, sicher auch in der Hoffnung, vom Verkauf von Anwendungen anderer Unternehmen profitieren zu können. Die APIs, die für die seinerzeit Windows 8 Store Apps genannten Anwendungen erforderlich waren, integrierte Microsoft direkt im Betriebssystem. Zwar ließen sich Anwendungen mit verschiedenen Programmiersprachen entwickeln (HTML/JavaScript, C++ und auch XAML), doch das .NET Framework blieb außen vor. Mit Windows 10 wurde das Konzept nochmals neu aufgerollt und wird seitdem unter dem Namen Universal Windows Platform Apps, kurz UWP-Apps, fortgeführt.

Mit WinUI hat Microsoft ein komplettes UI-Framework für UWP entwickelt, das insbesondere in Verbindung mit der Touch-Bedienung auf Tablets und Smartphones seine Vorzüge ausspielen kann. Einen Überblick erhält man in der XAML Controls Gallery, die sich aus dem App Store kostenlos installieren lässt. Der Code für das Beispielprogramm ist auf GitHub einsehbar [1]. Die App (Bild 1) demonstriert die verfügbaren Steuerelemente, die in vielen Fällen mit durchdacht gestalteten Animationen verknüpft sind.

Leider geht die Programmierung von UWP-Apps mit vielen Einschränkungen einher. Die Apps haben nur begrenzte Rechte, die der Anwender bei Bedarf zuordnen muss, die verwendbaren APIs sind sehr eingeschränkt (zum Beispiel kein .NET Framework) und die Funktionalität ist sehr eng mit der jeweiligen Windows-Version verknüpft. Daher können UWP-Programmierer auch nur bei den halbjährlichen Windows-Updates mit Neuerungen rechnen.



WinUI 2 XAML Controls Gallery für UWP, ein Beispielprogramm aus dem Microsoft App Store (Bild 1)

Aber auch auf Anwenderseite sind UWP-Apps nicht sonderlich beliebt. Windows-Anwendungen werden auch heute noch meist über Setup-Programme mit vollen Benutzerrechten eingerichtet, anders als unter macOS, iOS oder Android. Wer einmal den Windows Store aufgerufen hat, weiß, dass man dort kaum seriöse Anwendungen findet. Dubiose Spiele, die sich mit Werbung oder mit In-App-Verkäufen finanzieren, oder halb fertige, kaum nutzbare Anwendungen prägen das Erscheinungsbild des Stores. Die wenigen sinnvollen Apps gibt es zumeist auch als klassische Win32-Anwendung oder in Form einer Webseite. Da es auch kaum noch Smartphones gibt, auf denen UWP-Apps laufen, ist diese Plattform für App-Hersteller weitestgehend uninteressant geworden.

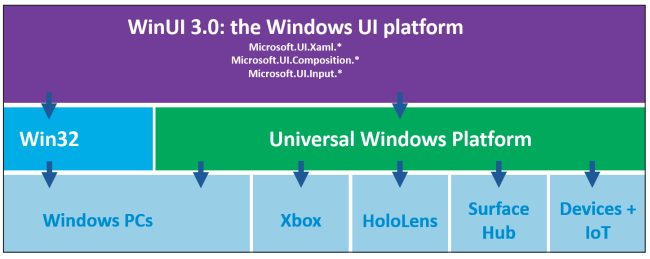


Bild: Microsoft [2]

WinUI 3 ist die native Benutzeroberflächenplattform-Komponente im Lieferumfang des Windows App SDK (Bild 2)

Vor ein paar Jahren hatte Microsoft dann die Idee, das UWP-Abstellgleis wieder mit der klassischen Win32-Programmierung zusammenzuführen [2] (Bild 2). Mit WinUI 3 sollte es mithilfe der bereits existierenden UI-Bibliothek möglich sein, sowohl Win32- als auch UWP-Anwendungen zu erstellen. Dazu musste allerdings die in Windows integrierte WinUI-2-Bibliothek aus dem Betriebssystem herausgelöst werden. WinUI 3 ist jedoch kein Stand-alone-Produkt, sondern integriert sich in das neue Windows App SDK [3], vormals als Preview unter dem Codenamen „Project Reunion" bekannt.

Während die Migration von WinUI 2 nach WinUI 3 für einen großen Teil der Komponenten vergleichsweise schnell vonstattenging, gestaltet sich die noch nicht abgeschlossene Implementierung des Windows App SDK deutlich zäher. Die Release-Version mit der Nummer 1.0 wurde erst Ende 2021 veröffentlicht (siehe Bild 3). Viele der ursprünglich geplanten Features sind aber noch längst nicht implementiert.

Interessanterweise fehlt bislang die Unterstützung für UWP (siehe Bild 4). Ob man das als Hinweis darauf werten kann, dass Microsoft UWP in Zukunft komplett fallen lassen will, bleibt offen. Microsoft empfiehlt jedenfalls den Entwicklern, die unbedingt bei UWP bleiben wollen, weiterhin WinUI 2 zu verwenden. Für neuere Anwendungen solle man auf WinUI 3 und die Win32-Plattform setzen. Infos zur WinUI-Roadmap und zur Migrationsstrategie finden Sie unter [4] und [5]. Eine tabellarische Übersicht der verfügbaren und geplanten Features zeigt Bild 5. Weitere Informationen zum Windows App SDK Release Channel finden Sie unter [6].

Auffällig und für viele Entwickler bestimmt auch besonders ärgerlich ist die Tatsache, dass insbesondere die Verknüpfung von WinUI 3 mit den bestehenden Technologien Windows Forms sowie Windows Presentation Foundation (WPF) bislang gar nicht umgesetzt wurde. Hierzu sollten die schon vor zwei Jahren vorgestellten XAML Islands dienen. Im Product Board unter [3] findet man diese Technik allerdings noch nicht einmal in der Rubrik *PLANNED*, sondern unter *UNDER CONSIDERATION* (siehe Bild 6).

Was geht bislang mit WinUI 3, und wie kann man es benutzen?

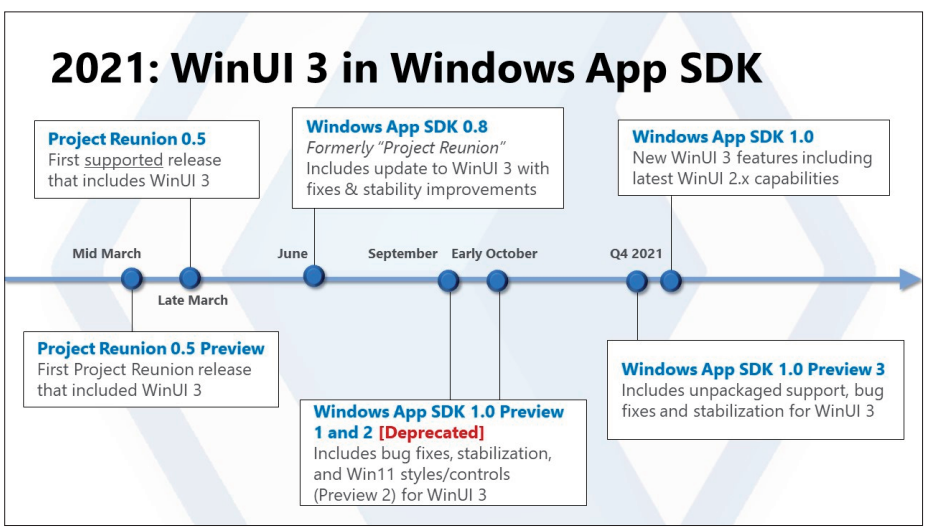
Einen ersten Eindruck von den verfügbaren Komponenten in WinUI 3 verschafft die Portierung der eingangs erwähnten XAML Controls Gallery. Diese finden Sie ebenfalls im Store unter dem Namen WinUI3 Controls Gallery (siehe Bild 7). Ein sehr großer Teil der Komponenten konnte bereits in die neue Welt überführt werden. Lediglich die Komponenten, die sich nicht so einfach aus dem Betriebssystem herauslösen ließen oder bei denen es andere Gründe gab, sind bislang nicht in der neuen Welt angekommen. Neu hinzugekommen ist aber mit *WebView2* das auf Chromium basierende Web-Browser-Control, das auch in WPF zur Verfügung steht und nun endlich eine zeitgemäße Integration von Webseiten in Win32-Anwendungen ermöglicht. Sie sollten die Gallery unbedingt einmal ausprobieren. Viele der Controls machen einen sehr hochwertigen Eindruck. Der Code der Gallery ist auf GitHub in einem anderen Branch der ursprünglichen WinUI 2 Gallery zu finden ([7]).

Natürlich können Sie auch Ihre eigenen Anwendungen mit WinUI 3 entwickeln. Den Einstieg hierzu beschreibt Microsoft unter [8] und [9]. Empfehlenswert ist die Verwendung von Visual Studio 2022 und .NET 6. In der genannten Dokumentation wird beschrieben, wie Sie das Windows App SDK installieren. Ferner ist eine Erweiterung für Visual Studio (VSIX-Datei) notwendig, damit die erforderlichen Templates in der Entwicklungsumgebung zur Verfügung gestellt werden können.

Nach erfolgreicher Installation stehen drei Vorlagen für WinUI-3-Projekte zur Verfügung (Bild 8):

- *Blank App, Packaged*, als einzelnes Projekt
- *Blank App, Packaged*, bestehend aus zwei Projekten
- *Class Library* (Klassenbibliothek mit vorbereiteten Verweisen zum Windows App SDK)

Vorlagen für Anwendungen ohne MSIX-Packages gibt es bislang nicht. Ein Projekt lässt sich aber manuell umstellen (siehe Anleitung unter [10]). Für einen ersten Start wählen Sie die erste Option aus der Vorlagenliste. Angelegt wird ein ▶



WinUI 3 unterstützt derzeit keine UWP-Apps (Bild 3)

fertig eingerichtetes Projekt mit einem Startfenster, auf dem mittig eine Schaltfläche platziert ist. Das Projekt lässt sich kompilieren und starten (kurioserweise manchmal erst im zweiten Anlauf).

Und gleich noch eine Auffälligkeit: Da fehlt doch was? Genau! Wo ist der Designer von Visual Studio?

Den gibt es bislang leider auch nicht. Sie müssen also mit XAML-Code vorliebnehmen. Für eingefleischte WPF-Programmierer ist das kein Problem, nutzen sie den Visual-Studio-Designer doch nur in Ausnahmefällen. Für Einsteiger könnte das jedoch die nächste Hürde sein, die zu überwinden ist. Zwar funktioniert ähnlich wie in WPF auch Edit & Continue, jedoch in der Regel nur, wenn sich die Änderungen auf den XAML-Code beschränken und nicht zu komplex sind. Man sollte sich auch nicht auf die Aktualisierung verlassen, denn manchmal verhält sich die Anwendung nach einem Neustart dann doch anders als nach der automatischen Aktualisierung (zum Beispiel beim Ändern von Datenbindungen).

Werfen wir einen Blick in die generierten Dateien der Anwendung. Das Hauptfenster (*MainWindow.xaml*, Listing 1) sieht dem einer WPF-Anwendung sehr ähnlich. Abweichend ist zum Beispiel die Syntax zum Einbinden von CLR-Namensräumen wie mit

```
xmlns:local="using:WinUI3Basics"
```

Die Klasse *Window* ist hier allerdings völlig anders implementiert worden. Während sie in WPF von der Klasse *ContentControl* abgeleitet wurde, erbt sie in WinUI 3 direkt von *System.Object* [11]. So fehlen diverse Eigenschaften wie zum Beispiel *Resources* oder *DataContext*. Die *Window*-Instanz selbst dient ebenfalls eher als Container für Ansichten (Pages). Die Instanzierung erfolgt auch nicht wie bei WPF im XAML-Code der App-Klasse, sondern in deren Code-behind-Datei *App.xaml.cs*:

```
protected override void OnLaunched(
    Microsoft.UI.Xaml.LaunchActivatedEventArgs args)
{
  m_window = new MainWindow();
  m_window.Activate();
}

private Window m_window;
```

Da es sich um eine Packaged App handelt, sorgt Visual Studio beim Starten der App automatisch für deren Installation. Sie können die App anschließend auch ohne Hilfe der Entwicklungsumgebung direkt aus Windows heraus starten.

Möchten Sie die App wieder loswerden, so müssen Sie sie manuell deinstallieren (Rechtsklick auf das App-Symbol, siehe Bild 9).

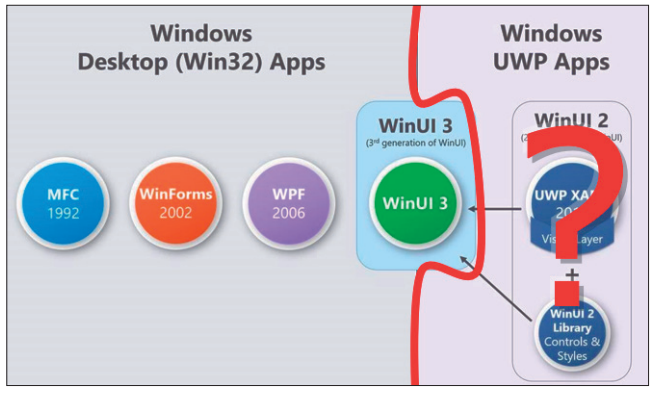


Bild: Microsoft [4]

Verlauf der Entwicklung des Windows App SDK (Bild 4)

Customer Capability	Project Reunion 0.5 (March 2021)	Windows App SDK 0.8 (June 2021)	Windows App SDK 1.0 (November 2021)	Planned for Windows App SDK 1.1	Planned for a future update
Supported in any app using the Windows App SDK	●	●	●	●	●
Contains new Windows 11 controls/styles from WinUI 2.6			●	●	●
Supports MSIX Deployment	●	●	●	●	●
Supports Unpackaged (non-MSIX) Deployment			●	●	●
Works downlevel to Windows 10 version 1809 and above	●	●	●	●	●
Supports the latest .NET	●	●	●	●	●
ARM64 support	●	●	●	●	●
<SwapChainPanel>	●	●	●	●	●
Intellisense, Hot Reload, Live Visual Tree	◆	◆	● [1]	●	●
Chromium-based WebView2	●	●	●	●	●
Title bar customization	●	●	●	●	●
Fluent Shadows	●	●	●	●	
Input validation for data fields	◆	◆	◆	◆	●
Supports multiple top-level windows on the same thread	◆	◆	◆	●	●
Support multiple top-level windows on separate threads	◆	◆	◆	◆	●
Drag and drop	●	●	●	●	●
RenderTargetBitmap	●	●	●	●	●
Mouse cursor customization	●	●	●	●	●
Animated Gif support	●	●	●	●	●
VirtualSurfaceImageSource (VSIS) support	●	●	●	●	●
In-app acrylic	●	●	●	●	●
Background acrylic				●	●
Mica				●	●
XAML Islands		◆	◆	◆	●
Media Controls (e.g. <MediaPlayerElement>)					●
<InkCanvas>					●
<MapControl>					■

Bild: Microsoft [4]

Verfügbarkeitsplan des Windows App SDK (Bild 5)

Ansichten aufbauen

Die Kapselung von Ansichten erfolgt bei WinUI 3 typischerweise mithilfe von Pages. Die Klasse *Page* ist abgeleitet von *UserControl*. Hier finden sich auch wieder die in WPF oft verwendeten Eigenschaften *DataContext* und *Resources*, die in ähnlicher Form benutzt werden können. Das Hinzufügen neuer Elemente erfolgt in Visual Studio wie üblich über das Kontextmenü des Projekts (*Add* | *New Item*, Bild 10). Neben der Page stehen weitere Elemente zur Auswahl, auf die später noch eingegangen wird.

Im einfachsten Fall kann eine Page als Unterelement des Window eingesetzt werden. Eine Komponente für die Navigation zwischen verschiedenen Pages ist aber bereits im Lieferumfang enthalten: *NavigationView*. Auch die Gallery benutzt sie, wie in den Abbildungen bereits zu sehen war.

Einen ersten Ansatz für die Implementierung einer Navigation mit drei Pages zeigt Listing 2. Die Navigationsstruktur

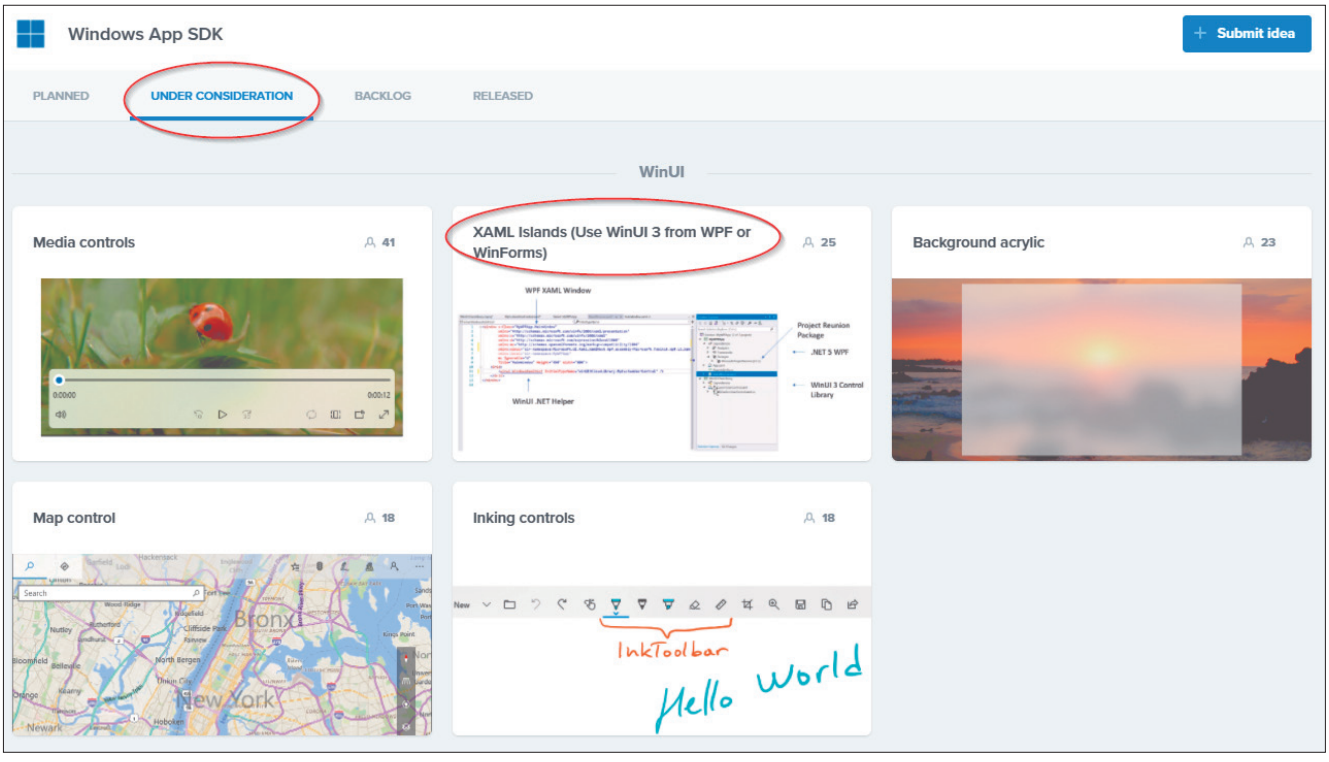


Bild: Microsoft [3]

Das kann noch dauern – nicht nur XAML Islands warten noch auf ihre Umsetzung (Bild 6)

ist hierbei fest im XAML-Code verdrahtet. Eine dynamische Lösung soll in einem späteren Beispiel folgen.

Als Unterelemente von *NavigationView* müssen zum einen die Menüeinträge und zum anderen ein Objekt vom Typ *Frame* als Platzhalter für eine anzuzeigende Page definiert werden. Leider funktioniert die Navigation nicht automatisch, vielmehr muss sie über den Event *ItemInvoked* zusätzlich implementiert werden.

Für jeden Eintrag des Navigationsmenüs werden drei Eigenschaften definiert. *Content* gibt den darzustellenden Inhalt des Menüpunkts an (hier der Anzeigetext). Über *Icon* kann ein Miniaturbild zugeordnet werden. Sie können selbst Icons definieren oder, wie hier, auf vorhandene zurückgreifen. Mit dem Setzen der *Tag*-Eigenschaft hat es hier eine besondere Bewandtnis, und die liegt in der etwas ungeschickten Umsetzung der Navigationsstruktur begründet. ▶

Viele (aber nicht alle) Komponenten haben den Weg von WinUI 2 nach WinUI 3 gefunden (Bild 7)

Listing 3 zeigt die Umsetzung des Eventhandlers für die eigentliche Navigation. Die Umschaltung zwischen den Pages erfolgt über die *Navigate*-Methode des *Frame*-Objekts. Und die ist leider sehr unglücklich implementiert worden. Als Parameter benötigt sie das Typ-Objekt der anzuzeigenden Page. Das macht den Umgang mit MVVM (Model-View-ViewModel)-Implementierungen und insbesondere den Einsatz von Dependency Injection unnötig kompliziert, da man in einem ViewModel in der Regel nichts mit Datentypen aus dem UI zu tun haben will. Hinzu kommt, dass die aus WPF bekannte Markup-Extension *{x:Type}* hier nicht zur Verfügung steht.

Somit bleibt als Workaround nur, die benötigten Typnamen im XAML-Code über die *Tag*-Eigenschaft festzulegen und im C#-Code hierüber das *Type*-Objekt zu ermitteln. Fehler im Typnamen führen dann konsequenterweise zu Exceptions bei der Ausführung.

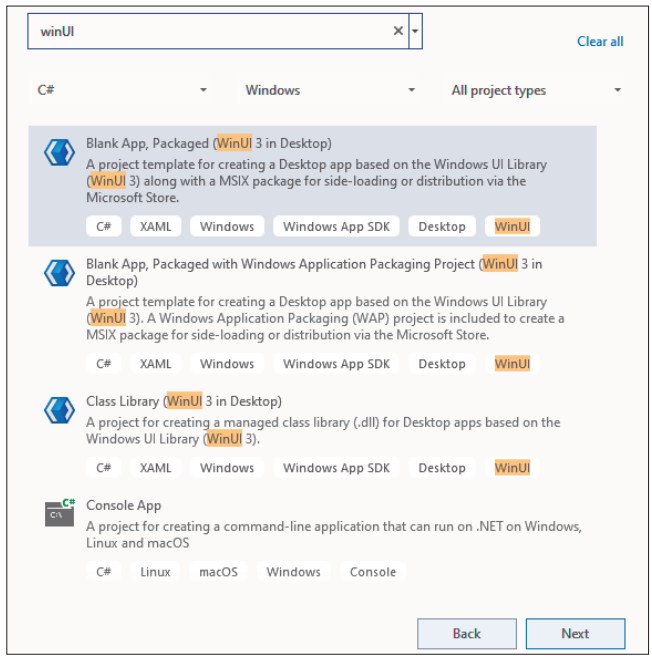
Das Navigationsfenster ist zweigeteilt. Links befindet sich das Navigationsmenü, rechts die ausgewählte Page. Verkleinert man das Fenster horizontal, werden unterhalb einer bestimmten Breite die Texte des Menüs ausgeblendet und es bleiben nur noch die Symbole sichtbar. Verkleinert man das Fenster noch weiter, werden auch diese ausgeblendet. Dann bleibt nur noch die Schaltfläche des Burger-Menüs (drei waagerechte Striche), die ohne weitere Positionierung allerdings mitten auf der Page angezeigt wird.

Die Schaltfläche für die Navigation zurück zur vorhergehenden Ansicht (Pfeil nach links) wird bereits angezeigt, hat aber noch keine Funktion. Auch das Zahnradsymbol, über das später ein Einstellungsdialog aufgerufen werden soll, ist bereits vorhanden. Ein Klick darauf führt aber bisweilen zum Absturz der App, da dieser Fall in der gezeigten Implementierung noch nicht berücksichtigt wird.

Listing 1: Das Hauptfenster des neuen Projekts

```
<Window
  x:Class="WinUI3Basics.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/
    xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/
    2006/xaml"
  xmlns:local="using:WinUI3Basics"
  xmlns:d="http://schemas.microsoft.com/expression/
    blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/
    markup-compatibility/2006"
  mc:Ignorable="d">

  <StackPanel Orientation="Horizontal"
      HorizontalAlignment="Center"
      VerticalAlignment="Center">
    <Button x:Name="myButton"
      Click="myButton_Click">Click Me</Button>
  </StackPanel>
</Window>
```



Visual-Studio-Vorlagen für WinUI 3 (Bild 8)

Unterschiede im Umgang mit Ressourcen und DataTemplates

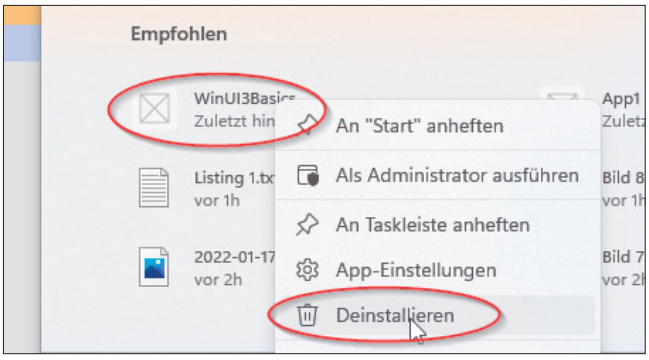
Grundsätzlich erfolgt das Anlegen von Ressourcen analog zur Vorgehensweise in WPF. Die *Application*-Klasse sowie die *Page*-Klasse besitzen die Eigenschaft *Resources*, über die eine Key-Value-Liste verknüpft ist. Sowohl der Key als auch der Value sind vom Typ *Object*, sodass beliebige Objekte referenziert werden können.

WinUI 3 kennt für den Abruf von Ressourcen nur die Markup-Extension *{StaticResource resourceKey}*. Die aus WPF bekannte Variante *{DynamicResource}* gibt es hier nicht. Das heißt, dass zum Zeitpunkt der Referenzierung einer Ressource diese auch existieren muss. Eine Aktualisierung beim späteren Austausch der referenzierten Objekte erfolgt nicht.

Auch bei den DataTemplates gibt es eine Einschränkung. In WPF ist es erlaubt, innerhalb einer Ressource ein DataTemplate ohne Angabe eines Keys zu definieren, wenn stattdessen die Eigenschaft *DataType* auf ein gültiges Typobjekt gesetzt wird. Bei jedem Vorkommen von Objekten des angegebenen Typs wird dieses DataTemplate dann automatisch anstelle der *ToString*-Implementierung greifen, sofern nicht explizit ein anderes Template angegeben wird. Das kann sehr praktisch sein, um zum Beispiel Beziehungen zwischen ViewModel und View zu definieren und eine automatische Anzeige einer View bei Vorgabe des zugehörigen ViewModels vorzusehen. In WinUI 3 geht das leider weder mit der klassischen Bindung noch mit der typisierten Variante.

Datenbindungen

Die klassische *{Binding}*-Markup-Extension steht, abgesehen von einigen Einschränkungen, auch für WinUI-3-Apps zur



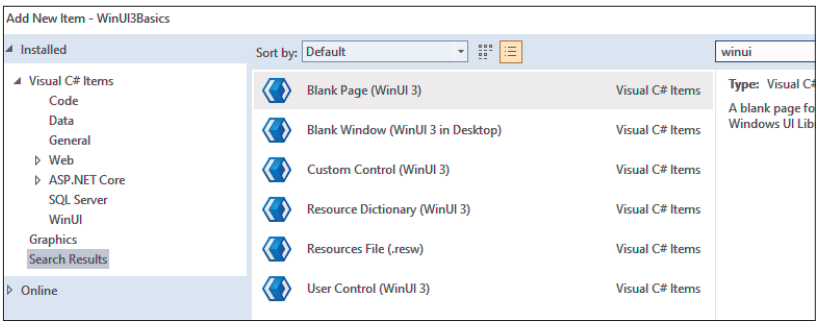
Manuelles Deinstallieren der App (Bild 9)

Verfügung. In den meisten Fällen wird man als Quelle für eine solche Datenbindung die *DataContext*-Eigenschaft einer Komponente heranziehen. Diese ist, wie bei WPF, in der Klasse *FrameworkElement* implementiert. Allerdings erben nicht alle relevanten Komponentenklassen von *FrameworkElement*. Wie bereits weiter oben erwähnt, gehört die Klasse *Window* nicht hierzu, die Klasse *Page* dagegen schon.

Bei der Vorgabe der Richtung, in der die Bindung aktiv sein soll (Eigenschaft *Mode*), stehen die Varianten *OneTime*, *OneWay* und *TwoWay* zur Verfügung. Die aus WPF bekannte Variante *OneWayToSource* gibt es hier nicht. Auch fehlt die Möglichkeit, eine Verzögerung für die Rückübertragung vom Ziel zur Quelle einzustellen (Eigenschaft *Delay* bei WPF). Schmerzlich vermissen wird allerdings mancher die Eigenschaft *StringFormat*, mittels derer man in WPF Textausgaben formatieren kann. Das lässt sich bei WinUI 3 nur auf Umwegen erreichen.

Was auch fehlt, ist die Möglichkeit, bei der Registrierung von Dependency-Properties über einen Parameter vom Typ *FrameworkPropertyMetadata* Metadaten-Einstellungen wie *BindsTwoWayByDefault* oder *Inherits* vorzunehmen. Die aus WPF bekannte Klasse gibt es bei WinUI 3 nicht. Mittels *BindsTwoWayByDefault* wird beispielsweise in WPF definiert, dass die *Text*-Eigenschaft einer TextBox als Default eine Zweiwegebindung aufbaut. Bei WinUI 3 muss man mit dem Typ *PropertyMetadata* vorliebnehmen, der lediglich die Definition eines Standardwerts und einer Rückrufmethode für den Fall von Wertänderungen vorsieht.

Neben der klassischen Datenbindung wartet WinUI 3 aber noch mit einer Compile-Zeit-Bindung auf, die bereits in UWP



Vorlagen für neue Elemente einer WinUI-3-App (Bild 10)

Listing 2: Einfache Navigation mit drei Pages

```
<Window
    ...>
  <NavigationView Name="nvSample"
      ItemInvoked="nvSample_ItemInvoked">
    <NavigationView.MenuItems>
      <NavigationViewItem Content="Basics"
        Icon="Crop" Tag="SomeBasics"/>
      <NavigationViewItem Content="Datenbindungen"
        Icon="Comment" Tag="Datenbindungen"/>
      <NavigationViewItem Content="Styles und
        Control-Templates" Icon="Stop"
        Tag="StylesUndControlTemplates"/>
    </NavigationView.MenuItems>
    <Frame Name="contentFrame"/>
  </NavigationView>
</Window>
```

unterstützt wurde. Hierzu dient die Markup-Extension *{x:Bind}*. Solche Bindungen werden bereits bei der Kompilation vollständig aufgelöst und können so die Performance verbessern, da zur Laufzeit keine aufwendigen Analysen der Bindungsausdrücke via Reflection mehr notwendig sind. Der Kontext für eine solche Bindung ist, sofern nicht anders festgelegt, die im Code-behind implementierte Klasse.

Listing 4 zeigt eine Beispielimplementierung für die Datenklassen *Artikel* und *Laden*. *Artikel* implementiert *INotifyPropertyChanged*, um über Änderungen der Eigenschaften informieren zu können (hier nur für die Eigenschaft *Preis* implementiert). Die Klasse *Laden* stellt eine Reihe von Artikeln in Form einer *ObservableCollection* bereit. Zusätzlich wird hier noch die Methode *ArtikelHinzufuegen* implementiert, die zu Demonstrationszwecken über *{x:Bind}* verknüpft werden soll.

Für die typisierte Bindung muss die Code-behind-Klasse der Page ergänzt werden (Listing 5). Die Eigenschaft *DerLaden* vom Typ *Laden* referenziert eine Instanz dieses Typs. Damit auch die klassischen Bindungen funktionieren, wird zusätzlich der *DataContext* gesetzt.

Listing 6 zeigt einige Beispiele für Datenbindungen, Bild 11 die Darstellung im Fenster. So lässt sich mittels klassischer Bindung mit

```
Text="{Binding Artikel[0].Bezeichnung}"
```

die *Text*-Eigenschaft eines TextBlocks mit der Bezeichnung des ersten Artikels aus der Liste verknüpfen. Der Bindungskontext ist in diesem Fall der *DataContext* der Page. Analog dazu funktioniert die kompilierte Bindung mit

```
Text="{x:Bind DerLaden.Artikel[0]
  .Bezeichnung}".
```

Listing 3: Für die Navigation ist das Typ-Objekt der anzuzeigenden Page notwendig

```
private void nvSample_ItemInvoked(NavigationView
    sender, NavigationViewItemInvokedEventArgs args)
{
  // Text aus Tag lesen
  string typename = args.InvokedItemContainer.Tag as
    string;

  // Typ-Objekt ermitteln
  typename = $"{nameof(WinUI3Basics)}.{typename}";
  var type = Assembly.GetExecutingAssembly()
    .GetType(typename);

  // Navigation anstoßen. Die Instanz der Page wird
  // automatisch über Reflection erzeugt
  contentFrame.Navigate(type);
}
```

Ausgangspunkt ist auch hier das *Page*-Objekt. Allerdings spielt hier der *DataContext* keine Rolle. Stattdessen wird direkt auf die Eigenschaften der *Page*-Klasse, in diesem Fall *DerLaden*, zugegriffen. Wegen der strengen Typisierung kann Visual Studio bereits bei der Eingabe im XAML-Code mit automatischer Code-Vervollständigung Unterstützung anbieten.

Auch bei *{x:Bind}* gibt es wieder die oben beschriebenen drei Varianten zur Festlegung der Bindungsrichtung über die Eigenschaft *Mode*. Allerdings gilt es hier eine Besonderheit

Listing 4: Klassische Datenklassen

```
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace WinUI3Basics
{
  public class Artikel : INotifyPropertyChanged
  {
    public int Artikelnummer { get; set; }
    public string Bezeichnung { get; set; }
    private double preis;

    public double Preis
    {
      get { return preis; }
      set
      {
        if (preis == value) return;
        preis = value;
        OnPropertyChanged();
      }
    }

    public event PropertyChangedEventHandler
      PropertyChanged;
    protected void OnPropertyChanged([
        CallerMemberName] string propertyName = null)
    {
      PropertyChanged?.Invoke(this,
        new(propertyName));
    }
  }

  public class Laden
  {
    public string Name { get; set; } = "Hinz & Kunz";
    public ObservableCollection<Artikel> Artikel {
        get; set; } = new()
    {
      new Artikel { Artikelnummer = 1, Bezeichnung =
        "Ball", Preis = 12 },
      new Artikel { Artikelnummer = 4, Bezeichnung =
        "Telefon", Preis = 49 },
      new Artikel { Artikelnummer = 6, Bezeichnung =
        "Lautsprecher", Preis = 23 },
      new Artikel { Artikelnummer = 71, Bezeichnung =
        "Tasse", Preis = 4 },
      new Artikel { Artikelnummer = 18, Bezeichnung =
        "Teller", Preis = 7 }
    };

    // Eventhandler, über {x:Bind} gebunden:
    // mit Parametern:
    // public void ArtikelHinzufuegen(object sender,
    // RoutedEventArgs e)
    // oder ohne:
    public void ArtikelHinzufuegen()
    {
      Artikel.Add(new Artikel { Artikelnummer=42,
        Bezeichnung="was neues", Preis=10 });
    }

  }
}
```

Listing 5: Typisierte Eigenschaft für die Bindung via {x:Bind}

```
public sealed partial class Datenbindungen : Page
{
  // Property für den typisierten Zugriff auf die
  // Daten
  public Laden DerLaden { get; set; }

  public Datenbindungen()
  {
    this.InitializeComponent();
    DerLaden= new Laden();

    // Setzen des DataContexts für klassische
    // Datenbindungen
    this.DataContext = DerLaden;
  }
}
```

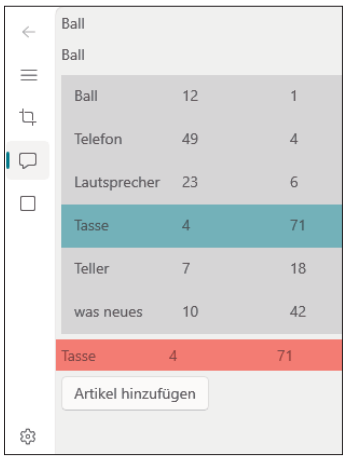
zu beachten: Der Standardwert ist hier, anders als bei der klassischen Bindung, nicht *OneWay*, sondern *OneTime*. Und das kann einen WPF-Programmierer wirklich an den Rand des Wahnsinns bringen!

Bindungen, die in WPF ohne Festlegung der *Mode*-Eigenschaft problemlos funktionieren würden, werden bei *{x:Bind}* einfach nicht aktualisiert. Das kostet unnötig viel Zeit bei der Fehlersuche. Im Beispiel muss für die *Content*-Eigenschaft des ContentControls, die an die Auswahl in der ListBox gebunden ist, *Mode* ausdrücklich auf *OneWay* gesetzt werden. Nur dann wirken sich Änderungen der Selektion der ListBox-Einträge auch auf die Anzeige auf dem ContentControl aus.

Microsoft begründet die Entscheidung, den Standardwert auf *OneTime* festzulegen, mit Performance-Verbesserungen. Klar, wenn die Bindung nur ein einziges Mal gemacht werden muss, kann auf jegliche Aktualisierungsmechanismen verzichtet werden. Praxisnah ist das jedoch nicht. Ein Ausweg ist hier, den Standard zumindest lokal außer Kraft zu setzen. In der Page kann über

```
x:DefaultBindMode="OneWay"
```

festgelegt werden, dass die Bindungsrichtung bei fehlender *Mode*-Angabe immer *OneWay* sein soll. Page-übergreifend in der *Window*-Klasse oder der *Application* lässt sich das aber anscheinend bislang nicht einstellen.



Beispiel für verschiedene Datenbindungen (Bild 11)

Bei der Definition von DataTemplates ist noch eine Besonderheit zu beachten, wenn man auch dort *{x:Bind}* einsetzen möchte. Wie in Listing 6 zu sehen ist, muss die Eigenschaft *x:DataType* gesetzt werden. Das ist notwendig, da der Compiler den Datentyp nicht selbst ableiten kann, ihn für die Bindung aber zwingend benötigt. Der Typ ist im Beispiel ja nicht die *Page*-Klasse, sondern die Klasse *Artikel*. Für klassische Bindungen, die zur Laufzeit aufgelöst werden, ist diese Angabe nicht erforderlich. Ein großer Vorteil von *{x:Bind}* ist hier sicherlich, dass der angegebene Pfad (Eigenschaft *Path* beziehungsweise Konstruktorparameter der Markup-Extension) bereits zur Compile-Zeit stimmen muss, da der Compiler sonst mit einer Fehlermeldung abbricht. Bei der klassischen Bindung wird ein Fehler bei der Definition des Bindungspfads erst zur Laufzeit erkannt.

Eine andere Möglichkeit, die *{x:Bind}* zu bieten hat, stellt allerdings einen großen Vorteil gegenüber der klassischen Bindung dar: Mittels *{x:Bind}* lassen sich Events mit Methoden verknüpfen. Die Eventhandler müssten normalerweise im Code-behind der *Page*-Klasse implementiert werden. In WPF muss man entsprechende Umwege beschreiten, wenn man die Aktion an ein ViewModel delegieren möchte (Commands, Behaviors et cetera). In WinUI 3 kann der Eventhandler einfach über *{x:Bind}* festgelegt werden (siehe Listing 6):

```
<Button Content="Artikel hinzufügen"
  Click="{x:Bind
  DerLaden.ArtikelHinzufuegen}" />
```

Die zugehörige Methode ist hier in der Klasse *Laden* definiert. Bei der Definition der Methode hat man die Wahl, ob man diese ohne Parameter deklariert oder ob man die Signatur des Eventhandler-Delegates umsetzt und alle erforderlichen Parameter vorsieht. Letzteres wird in vielen Fällen bei MVVM vermutlich wenig hilfreich sein, da ein ViewModel die View nicht kennt und somit nichts mit Informationen wie zum Beispiel der Mausposition anfangen kann.

Ein Hinweis noch: Beim Erstellen der Demo-Anwendung kam es zu kuriosen Fehlermeldungen, wenn im Namen der Methode Umlaute verwendet wurden. Hier scheinen noch ein paar Bugs zu schlummern.

Wie geht es weiter?

In den folgenden Artikeln wollen wir untersuchen, wie sich mit WinUI 3 eine moderne Desktop-Anwendung erstellen lässt, die auf MVVM und Dependency Injection beruht. Microsofts Demos wie die Controls Gallery zeigen zwar sehr schön die vielfältigen und funktional teilweise sehr üppig ▶

Listing 6: Datenbindungen mit klassischer Bindung sowie mit {x:Bind}

```
<Page
    x:Class="WinUI3Basics.Datenbindungen"
    ...>

  <Page.Resources>
    <DataTemplate x:DataType="local:Artikel"
        x:Key="ArtikelTemplate">
      <StackPanel Orientation="Horizontal">
        <TextBlock Text="{x:Bind Bezeichnung}"
          Width="100"/>
        <TextBlock Text="{x:Bind Preis}"
          Width="100"/>
        <TextBlock Text="{x:Bind Artikelnummer}"
          Width="100"/>
      </StackPanel>
    </DataTemplate>
  </Page.Resources>

  <StackPanel>
    <TextBlock Text="{Binding Artikel[0].Bezeichnung}"
      Margin="5"/>
    <TextBlock Text="{x:Bind
      DerLaden.Artikel[0].Bezeichnung}"
      Margin="5"></TextBlock>
    <ListBox ItemsSource="{x:Bind DerLaden.Artikel}"
             Margin="5" Name="LB"
           ItemTemplate="{StaticResource
             ArtikelTemplate}"
           Background="{StaticResource
             SystemRevealListLowColor}"/>

    <Border Background="Salmon">
      <ContentControl Content="{x:Bind
                       LB.SelectedItem,
                       Mode=OneWay}" Margin="5"
                     ContentTemplate="{StaticResource
                       ArtikelTemplate}"/>
    </Border>

    <Button Content="Artikel hinzufügen"
      Click="{x:Bind DerLaden.ArtikelHinzufuegen}"
      Margin="5"/>
  </StackPanel>
</Page>
```

ausgestatteten Steuerelemente, für den Praxiseinsatz taugen diese Beispiele aber nur bedingt. Vieles wird im XAML-Code fest verdrahtet, wohingegen man in der Praxis oftmals mehr Flexibilität benötigt. Der eine oder andere Trick kann hier weiterhelfen. Auch unerwartete Abstürze werden noch ein Thema sein. Denn leider ist so manche Fehlermeldung, die den Tiefen der C++/COM-Programmierung des Windows App SDK entspringt, wenig hilfreich.

Ein weiterer Schwerpunkt wird der Einsatz von Steuerelementen sein, die es bei WPF nicht „out of the box“ gibt. Controls wie GridView, FlipView oder SemanticZoom können das Erscheinungsbild einer Desktop-Anwendung deutlich verbessern.

Weitere Informationen zu WinUI 3 finden Sie unter [12], [13] und [14]. Der Sourcecode der gezeigten Beispiele ist auf GitHub unter [15] einsehbar. ■

[1] XAML Controls Gallery auf GitHub (WinUI 2), www.dotnetpro.de/SL2204WinUI3_1

[2] Windows-UI-Bibliothek (WinUI) 3, www.dotnetpro.de/SL2204WinUI3_2

[3] Windows App SDK Product Board, www.dotnetpro.de/SL2204WinUI3_3

[4] Windows UI Library Roadmap, www.dotnetpro.de/SL2204WinUI3_4

[5] Overall migration strategy, www.dotnetpro.de/SL2204WinUI3_5

[6] Windows App SDK release channels, www.dotnetpro.de/SL2204WinUI3_6

[7] WinUI 3 XAML Controls Gallery, www.dotnetpro.de/SL2204WinUI3_7

[8] Overview of app development options, www.dotnetpro.de/SL2204WinUI3_8

[9] Create a WinUI 3 app, www.dotnetpro.de/SL2204WinUI3_9

[10] Advanced tutorial: Build and deploy an unpackaged app that uses the Windows App SDK, www.dotnetpro.de/SL2204WinUI3_10

[11] API-Dokumentation, www.dotnetpro.de/SL2204WinUI3_11

[12] Windows UI Library (WinUI), www.dotnetpro.de/SL2204WinUI3_12

[13] About WinUI, www.dotnetpro.de/SL2204WinUI3_13

[14] Windows UI Library auf GitHub, www.dotnetpro.de/SL2204WinUI3_14

[15] Beispielcode zum Artikel, www.dotnetpro.de/SL2204WinUI3_15

Joachim Fuchs
ist begeisterter Anhänger von Microsofts .NET-Philosophie. Er arbeitet als Softwarearchitekt, Berater und Dozent im Expertennetzwerk www.it-visions.de. Seine Schwerpunkte liegen derzeit bei XAML- und Web-UI-Technologien.
dnp@fuechse-online.de

dnpCode A2204WinUI3