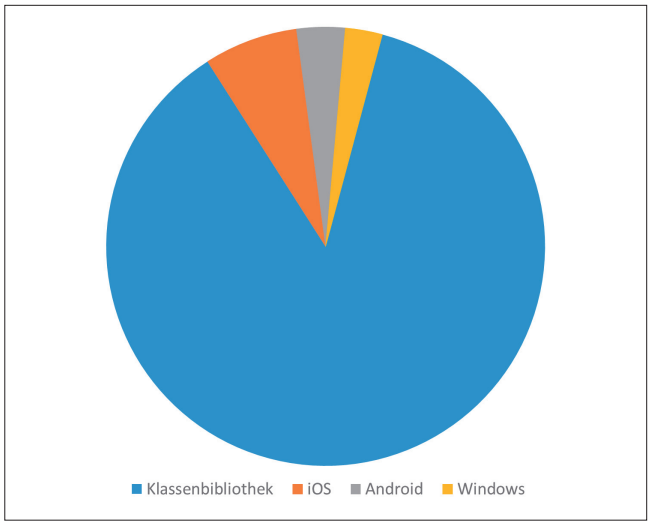
NATIVE PLATTFORMÜBERGREIFENDE APPS

Apps entwickeln? Xamarin!

Eine Applikation programmieren, die alle Nutzer erreicht, eine einheitliche Oberfläche hat und Kosten spart.

Bei der App-Entwicklung stellt sich immer die Frage: nativ oder plattformunabhängig? Der Vorteil von Letzterem liegt auf der Hand. Zwar lassen sich plattformspezifische Apps zum Beispiel für iOS mit Apples Entwicklungsumgebung Xcode entwickeln und mit allem ausstatten, was das SDK so bietet: Unterstützung für 3D-Touch, Gesten, Animationen, Dateiverschlüsselung und diverse Spielereien. Doch diese App danach für andere Plattformen zu portieren ist aufwendig und benötigt bei entsprechend komplexen Apps ein separates Entwicklerteam. Wenn verschiedene Entwicklerteams die gleiche App für iOS, Android und Windows parallel entwickeln, dann kann es aufgrund von unterschiedlichen Programmiersprachen, anderen Vorstellungen oder Richtlinien zusätzlich zu Problemen kommen. Je nach Programmiersprache und Entwicklungsmuster ist die Wiederverwendbarkeit des Quelltexts dann nur eingeschränkt gegeben.

Um gar nicht erst vor diesem Dilemma zu stehen, lohnt es sich, schon vorab über plattformübergreifende Entwicklungslösungen nachzudenken. Entweder man entscheidet sich für hybride Lösungen wie Ionic oder Apache Cordova, die denselben Webinhalt auf jeder Plattform anzeigen, oder man wählt native Lösungen, wie React Native, NativeScript oder Xamarin, die „richtige“ Apps erzeugen. Die nativen Frameworks sind dabei schneller und sehr viel hardwarenäher als die hybriden Webframeworks und geben dem Nutzer eine Benutzeroberfläche, wie sie es von ihrem Betriebssystem gewohnt sind.



Code-Verteilung, gemessen an der Menge des Quelltexts (Bild 1)

Die Herausforderung

Das Ziel des Projekts, das diesem Artikel zugrunde liegt, war es, die elementaren Funktionen einer großen Webanwendung mit REST-Schnittstelle nativ mithilfe von Xamarin auf mobile Geräte zu bringen.

Der Server, der die Weboberfläche bereitstellt, ist eine Reporting-Software, die Berichte aus beliebigen Datenbanken visualisieren kann. Berichtsvorlagen werden mit einer separaten Software oder im Ad-hoc-Designer erstellt. Dort können die gewünschten Daten mit Diagrammen, Tabellen oder Graphen verknüpft werden. Ein Bericht, der zum Beispiel die Verkaufszahlen eines Unternehmens in einem Balkendiagramm darstellt, lässt sich dann mit den aktuellen Daten aus der Datenbank in beliebige Dateiformate exportieren. Mit mobilen Apps ist dieser Zugriff dann auch bequem von unterwegs aus möglich.

Xamarin

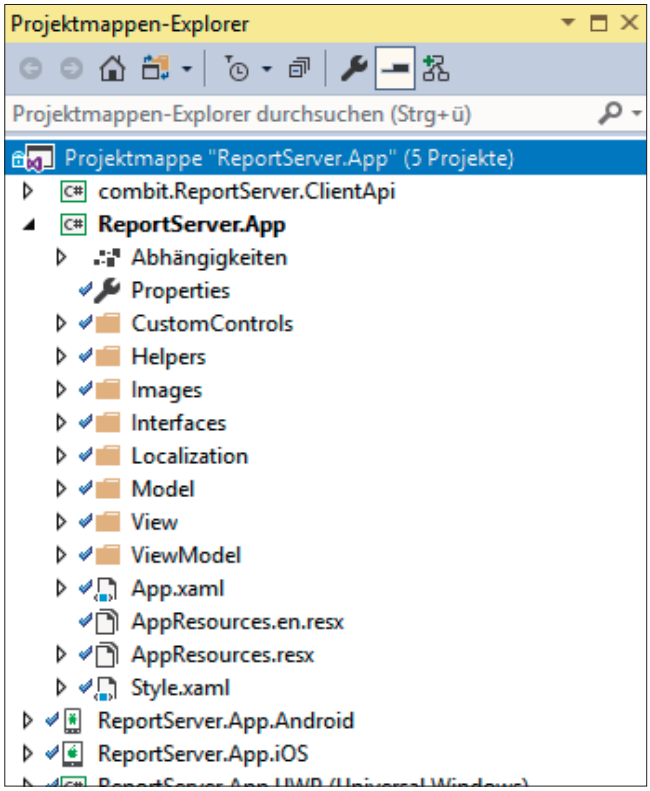
Xamarin ist in Visual Studio integriert, seit Microsoft Xamarin übernommen hat, was das frühere Xamarin Studio obsolet macht. Zudem entstehen – im Unterschied zu früher – auch keine weiteren Kosten bei der Lizenzierung [1]. So kann man mit aktuellen Visual-Studio-Versionen direkt anfangen, auch kommerzielle Xamarin-Apps zu schreiben.

Xamarin-Projekte haben in der Regel eine gemeinsame C#-Codebasis, aus der eine echte native App für jede Plattform erstellt wird. Neben Android und iOS unterstützt Xamarin auch Windows. Weitere Plattformen wie Tizen, macOS, Linux (GTK) und WPF werden ständig ergänzt [2].

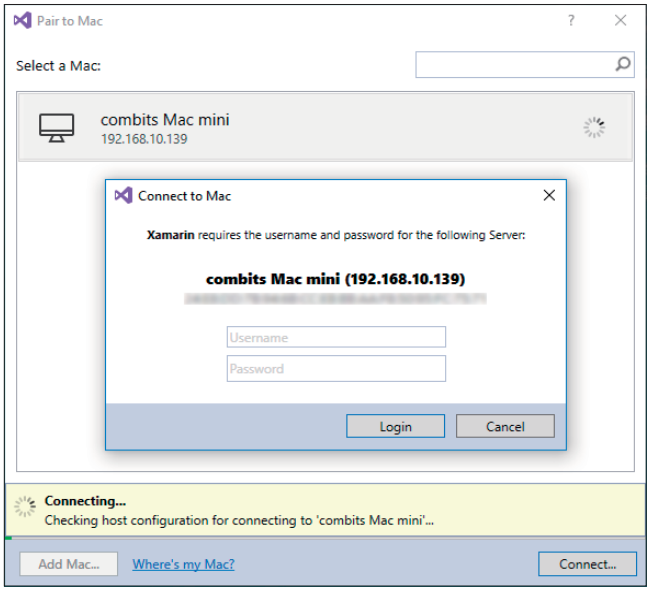
Die Vorteile gegenüber der nativen Entwicklung separater Applikationen sind die geringere Redundanz des Codes, eine gemeinsame Programmiersprache und vor allem die Verwendung einer großen geteilten Bibliothek für Benutzeroberfläche und Logik. Durch einen Code-Reuse von bis zu 100 Prozent lässt sich Zeit und damit auch Geld sparen.

In der Grafik von Bild 1 wurde ausschließlich die Größe des in unserem Beispielprojekt hinzugefügten Quellcodes gemessen. Abhängig von der Komplexität und Detailliertheit der App ist es in der Regel nicht möglich, 100 Prozent Wiederverwendung zu erreichen.

Die Grundlage für Xamarin ist dabei das altbekannte Mono Framework. Mit Mono ist es schon lange möglich, Microsoft-.NET-Programme auch ohne Neuübersetzung auf UNIX- beziehungsweise Linux- Betriebssystemen auszuführen. Dadurch kann man Apps in C# schreiben, die dann auf von Mono unterstützten Plattformen laufen.



Die Struktur der Solution (Bild 2)



Herstellen der Verbindung mit einem Mac (Bild 3)

Das Gerüst

Eine .NET-Standard-Klassenbibliothek stellt den gemeinsamen Code bereit, der von allen Plattformen geteilt wird. Für jede Plattform existiert dann neben der Klassenbibliothek ein Projekt in der Solution.

Um die Webanwendung anzusprechen, wird ein weiteres C#-Projekt eingebunden, das die Kommunikation mit dem REST API des Servers übernimmt (Bild 2).

Nun kommt Xamarin.Forms ins Spiel, das UI-Framework zur eigentlichen plattformübergreifenden Entwicklung, mit dem nahezu 100 Prozent Code-Reuse möglich sind [3]. Es stellt Steuerungselemente und Mechanismen zur Verfügung, um die gemeinsame Benutzeroberfläche für die verschiedenen Zielplattformen zu erstellen. Verfügbar ist es als NuGet-Paket, das häufig aktualisiert wird und von dem auch Vorveröffentlichungen bereitgestellt werden, um Bugfixes schnell verfügbar zu machen. Neben normalen NuGet-Paketen für alle .NET-Projekte gibt es auch speziell für Xamarin entwickelte Bibliotheken, Steuerungselemente und Android-Support-Bibliotheken.

Beim Erstellen des jeweiligen Projekts ist die Plattform auszuwählen, für welche die App erstellt werden soll. Für Android ist es ein Application Package Kit (Dateierweiterung *.apk*), für Windows eine ausführbare Datei (*.exe*) und für Apples Mobilbetriebssystem ein iOS App Store Package (*.ipa*).

Da Xcode beziehungsweise der Apple Compiler nicht für Windows verfügbar ist, kann man die iOS-App nicht auf einem Windows-Computer erstellen. Um iOS-Apps mit Xamarin zu erstellen, benötigt man einen Mac, auf dem Visual Studio für Mac und Xcode installiert sind. Entweder man entwickelt direkt auf dem Mac, oder man benutzt ihn als „Agent" von einem Windows-Computer aus. Dazu verwendet man einfach die Adresse des Macs und gibt Benutzername und Passwort ein (Bild 3).

Der Quellcode wird dann zum Mac übertragen, dort kompiliert und auf dem angeschlossenen Gerät installiert. Außerdem kann die Ausgabe von iOS-Simulatoren, die auf dem Mac laufen, an den Windows-PC übertragen werden.

Entwurf der Benutzeroberfläche

Wer auf einen Designer für die Benutzeroberfläche bei Xamarin hofft, wird enttäuscht. Hier ist im Moment noch XAML-Handarbeit gefragt. Microsoft hat mit den letzten Visual-Studio-Patches in diesem Bereich Erweiterungen vorgenommen, derzeit ist aber das direkte XAML-Coding noch die Methode der Wahl. Dank des separaten Vorschaufensters ist das keine große Hürde, die Vorteile von XAML überzeugen schnell. Zum einen liest es sich leichter als komplett programmatisch erzeugte Oberflächen in C#, und zum anderen können Konflikte bei der Versionierung einfacher gelöst werden als in designergeneriertem Code.

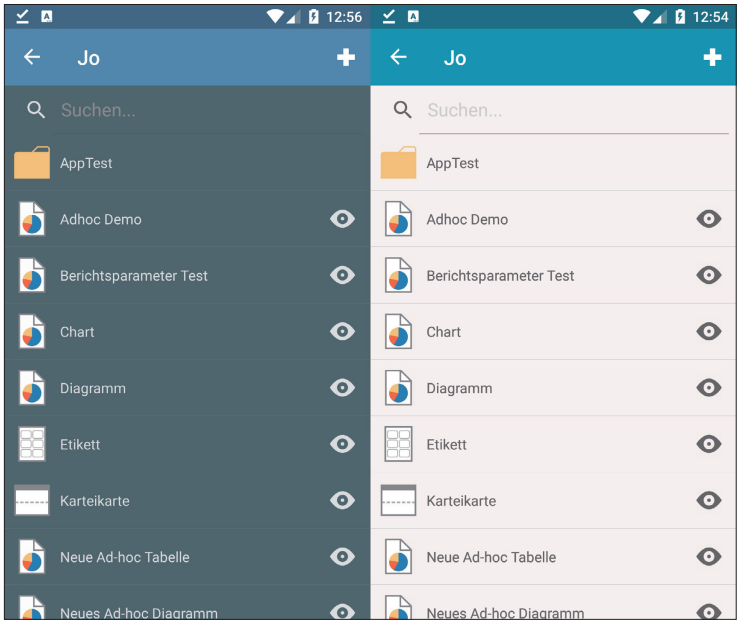
In XAML stehen verschiedenste Seiten, Layouts, Zellen und viele andere Steuerungselemente zur Verfügung [4]. Farben und auch ganze Themes können dynamisch geladen werden, sodass zur Laufzeit der App das Aussehen geändert werden kann. Sobald sich der Farbwert einer Ressource ändert, aktualisiert sich jede Komponente, die diese Farbressource benutzt. Das ermöglicht mit wenig Aufwand einen Nachtmodus oder eine generelle Theme-Auswahl (Bild 4).

Man kann in XAML, abhängig von der Plattform, unterschiedliche Abstände, Farben, Sichtbarkeiten und andere verfügbare Eigenschaften setzen. Falls aber Bedingungen benötigt werden oder Zellen einer Tabelle vom Inhalt abhängig sein sollen, muss eine separate C#-Klasse erstellt werden.

Jedes Xamarin.Forms-Steuerungselement wird von einem Renderer in die nativen Komponenten von Android, iOS ▶

und Windows umgewandelt. Diese Renderer kann man als Entwickler erweitern und so den Komponenten ein benutzerdefiniertes Aussehen und Verhalten verleihen. Damit die App einheitlich aussieht, kommt man aufgrund der verschiedenen nativen Komponenten und Richtlinien kaum um eigene Renderer herum.

Die ListView ist eine der wichtigsten Komponenten, da sie beliebig viele Elemente tabellarisch darstellen kann. Das Steuerungselement wird in eine UITableView auf iOS umgewandelt. Im Renderer zum Beispiel im iOS-Projekt erbt man dann von der Klasse *ListViewRenderer* und überschreibt nach Bedarf die Methode *OnElementChanged*. Dort hat man dann durch die Eigenschaft *Element* Zugriff auf das Xamarin-Element (ListView), und durch die Eigenschaft *Control* kann die gerade erstellte iOS-Komponente (UITableView) angesprochen werden (Listing 1). So lassen sich Änderungen direkt am nativen Steuerungselement vornehmen (Bild 5). Konvention ist es, vom Xamarin-Steuerungselement zu erben und dann explizit nur diese Subklasse zu rendern. Man erstellt beispielsweise eine Klasse *AdvancedListView*, kann in XAML die selbst definierten Eigenschaften setzen und in den Renderern auf jeder Plattform auf diese zugreifen und basierend darauf das Element verändern.



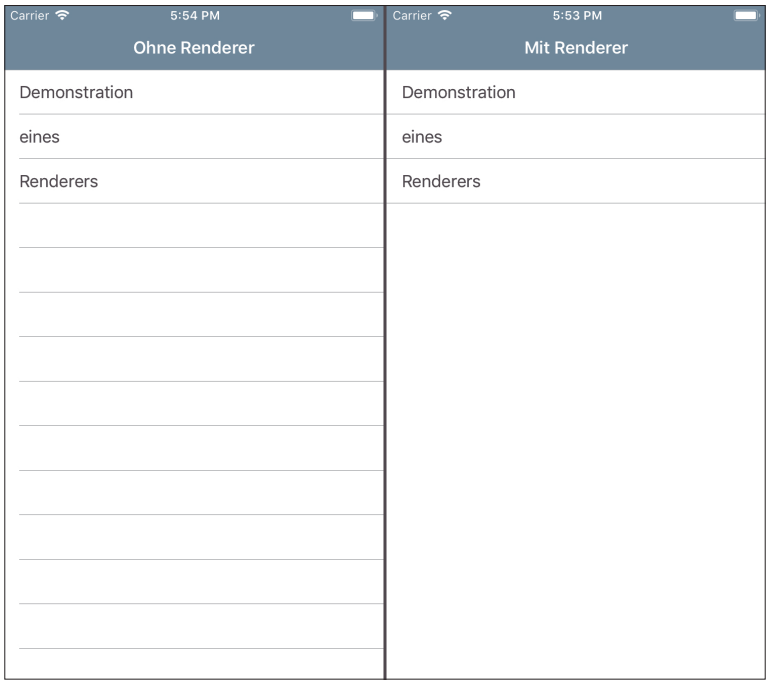
Vergleich zweier Farbschemata auf Android (Bild 4)

Was jede größere Applikation benötigt, ist ein Menü. Dessen wohl am weitesten verbreitete Ausprägung ist das Hamburger-Menü (Bild 6). Bei einem Klick auf das sogenannte Hamburger-Icon erscheint eine Seite, die über die aktuelle Ansicht gelegt wird oder die aktuelle Seite beiseiteschiebt. Bei Xamarin lässt sich einfach eine MasterDetailPage verwenden, an welche die Menü-Seite und die aktuelle Seite angehängt werden. Zusätzlich kann dann das Verhalten beziehungsweise die Animation des Menüs gewählt werden. Das ist ein sehr praktisches Feature und kaum irgendwo anders so einfach zu implementieren.

Das Entwicklungsmuster

Das Entwicklungsmuster Model-View-ViewModel, kurz MVVM, ist eine Variante des Model-View-Controller-Musters mit besserer Testbarkeit und geringerem Implementierungsaufwand. Wie MVC trennt es die Präsentation von der Logik. MVVM wird besonders bei WPF-Applikationen eingesetzt, findet aber beispielweise auch in Cocoa, JavaFX und HTML5 Verwendung.

Die Benutzeroberfläche (View) ist direkt an die Eigenschaften des ViewModels gebunden. Sobald sich eine Property im ViewModel ändert, wird im benutzerdefinierten Setter der Eigenschaft eine Benachrichtigung ausgelöst, die eine



Beispiel-Renderer für eine ListView (Bild 5)

Das Hamburger-Menü der Mobilapplikationen (Bild 6)

Listing 1: Renderer für eine ListView auf iOS

```
using UIKit;
using Xamarin.Forms;
using Xamarin.Forms.Platform.iOS;
using ReportServer.App.iOS.Renderers;

// Registriert diesen Renderer für alle
// Steuerungselemente des Typs "ListView"
[assembly: ExportRenderer(typeof(ListView),
  typeof(CustomListViewRenderer))]

namespace ReportServer.App.iOS.Renderers
{
  public class CustomListViewRenderer :
    ListViewRenderer
  {
    protected override void OnElementChanged(
      ElementChangedEventArgs<ListView> e)
    {
      base.OnElementChanged(e);
      // this.Element : Xamarin.Forms.ListView
      // this.Control : UIKit.UITableView

      // Sofern das iOS-Steuerungselement bereits
      // erstellt ist
      if (Control != null)
      {
        // Entfernt alle unnötigen Trennlinien, indem
        // eine (leere) TableFooterView hinzugefügt
        // wird
        Control.TableFooterView = new UIView();
        // Setzt den Abstand der Trennlinien zu allen
        // Rändern auf 0
        Control.SeparatorInset = UIEdgeInsets.Zero;
      }
    }
  }

}
```

Änderung induziert. Im Endeffekt bezieht das ViewModel die Informationen aus der Geschäftslogik (Model) und setzt dementsprechend seine Properties, die automatisch in der View präsentiert und aktualisiert werden (Bild 7).

Es gibt Dutzende MVVM-Frameworks für Xamarin, die den Aufwand für die Implementierung des Entwurfsmusters reduzieren. Der Vorteil an diesem Pattern ist, dass der Programmcode beziehungsweise die Oberfläche sehr leicht und schnell anzupassen sind und man die Aufgaben gut im Team aufteilen kann. Beispielsweise können die Designer unabhängig von den Programmierern an der View arbeiten. Besonders hilfreich ist es für Entwickler, die die Applikation übernehmen oder erweitern, da der Code sehr übersichtlich ist.

Fazit

Lohnt sich der Einstieg in Xamarin, wenn man bisher nativ unterwegs ist oder neu in die mobile Entwicklung einsteigen möchte? Wenn man alle Plattformen unterstützen will – definitiv ja. Die Einrichtung des Macs benötigt zwar zusätzlich Zeit, falls man auf einem Windows-Computer entwickelt, das ist aber eine einmalige Aktion.

Die Umstellung auf XAML ist reine Gewöhnungssache und im Zusammenspiel mit dem MVVM-Pattern eine echte Bereicherung. Zu Beginn ist der Entwurf wegen der teilweise fehlenden Vorschau und langer Kompilierzeiten langsamer als mit einem grafischen Designer, aber dank der Visual-Studio-Updates gibt es inzwischen mehr automatische Vervollständigungen und Hilfsaktionen, die das Designen leichter machen.

Da man um plattformspezifischen Code und Konfigurationsdateien selten komplett herumkommt, muss man sich mit den Grundlagen jeder Plattform auseinandersetzen. Deshalb ist es bei komplexeren Apps weiterhin hilfreich, ein komplettes Entwicklerteam zu haben – im Idealfall mit einem Spezialisten für jedes Betriebssystem, denn jedes hat seine Eigenheiten und Einschränkungen.

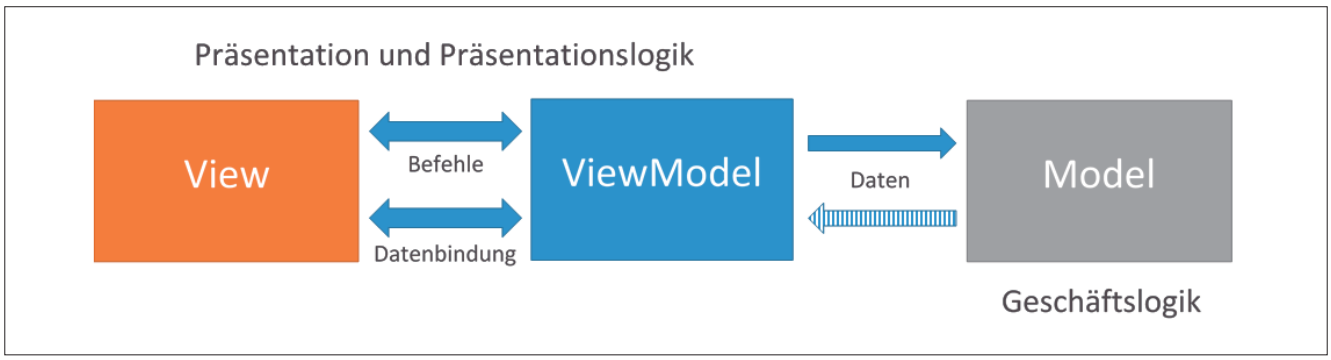
NuGet-Pakete sind bei der Entwicklung unumgänglich, allerdings auch ein zweischneidiges Schwert. Es gibt viele Pakete, die man relativ schnell benötigt, da Xamarin.Forms auf die essenziellen Steuerungselemente beschränkt ist. Bei manchen Paketen gibt es allerdings Kompatibilitätsprobleme zwischen den .NET Frameworks oder einfach Bugs, da die Erweiterungen teilweise noch nicht ausgereift sind. Auch im stabilen Kanal von Xamarin sind schon Bugs aufgetreten, welche die Funktionalität auf einer Plattform so eingeschränkt haben, dass ein Zurückkehren zu einer älteren Version nötig war. Das zeigt, wie stark man von Xamarin und der doch recht komplexen Erstellprozedur abhängig ist. Dank der vielen aktiven Maintainer auf GitHub werden Bugs aber in aller Regel schnell bearbeitet.

Die finale App ist auf den Plattformen durch Xamarin beziehungsweise Mono etwas größer als „echte" native Apps. Bei UWP-Apps wird sogar ein Installer gebaut, was sehr hilfreich für das Veröffentlichen ist.

Mit Xamarin lassen sich in sehr kurzer Zeit Menüs und Ansichten aufbauen (Bild 6). Außerdem können plattformspezifische Besonderheiten, zum Beispiel Widgets und UI-Feinarbeiten wie Animationen, auf jeder Plattform individuell implementiert werden.

Wenn man aus der .NET-Entwicklung kommt, ist der Einstieg in Xamarin leicht, und man kann bestehende Projekte mit Business-Logik direkt in Apps einbinden. Visual Studio und auch XAML können dann wie gewohnt verwendet werden.

Auch bestehende Apps zu ersetzen, wie dies etwa der ▶



Das MVVM-Konzept im Überblick (Bild 7)

Paketdienstleister UPS bereits getan hat [5], kann sich lohnen. Nach Abschluss des Grundgerüsts sind Features sehr schnell zu implementieren. ■

[1] Xamarin for Everyone, https://blog.xamarin.com/xamarin-for-all
[2] Glimpse the Future of Xamarin Forms 3.0, www.dotnetpro.de/SL1810CrossApp1
[3] Veikko Krypczyk, Simultan: Native Apps mit Xamarin, dotnetpro 7/2018, S. 118 ff., www.dotnetpro.de/A1807AppXamarin
[4] Xamarin-Steuerelement-Referenz, www.dotnetpro.de/SL1810CrossApp2
[5] Microsoft Customer Stories: UPS, www.dotnetpro.de/SL1810CrossApp3

Joshua Rutschmann
studiert Angewandte Informatik an der HTWG Konstanz und absolviert derzeit ein Praktikum bei combit. Er ist früh in die Welt des Programmierens eingestiegen: von Visual Basic über C#, Objective-C bis hin zu Swift.
joshua@rutschmann.tech

dnpCode A1810CrossApp