HANSER

Leseprobe

zu

Cross-Plattform-Apps mit Xamarin.Forms entwickeln

von Krämer André

Print-ISBN: 978-3-446-45155-1
E-Book-ISBN: 978-3-446-45366-1

Weitere Informationen und Bestellungen unter

https://www.hanser-kundencenter.de/fachbuch/artikel/9783446451551

sowie im Buchhandel

Inhalt

Vorwort

Im Frühjahr 2013 hörte ich auf der Fahrt zu einem Kunden die Tablet Show, einen früheren Podcast von Carl Franklin und Richard Campbell zum Thema Mobile Development. Mein Hauptinteresse galt damals, wie bei so vielen .NET Entwicklern, der Entwicklung von Windows-8-Apps. Gemeinsam mit einem Bekannten hatte ich wenige Monate zuvor eine App für Windows 8 veröffentlicht, die zu dieser Zeit gerade die Schallmauer von 1 000 000 Downloads durchbrach.

In früheren Episoden der Tablet Show hatte ich bereits einige nützliche Tipps erhalten, die uns bei der Optimierung unserer App halfen, und so war ich sehr neugierig, was diese Folge Neues bringen würde.

In dieser Folge berichtete Miguel de Icaza, einer der beiden Gründer der Firma Xamarin, dass es eine kostenfreie, eingeschränkte Einsteigerversion von Xamarin geben wird sowie eine günstige Variante für einzelne Entwickler. Außerdem kündigte er an, dass die Entwicklung von Xamarin-Apps von nun an auch in Visual Studio möglich war.

Ab diesem Zeitpunkt war ich von Xamarin fasziniert. Die Tatsache, dass ich mit meinen C#-, .NET- und Visual-Studio-Kenntnissen dank Xamarin in der Lage war, Apps für Android und iOS zu schreiben, ohne Java oder Objective-C lernen zu müssen, begeisterte mich, und diese Begeisterung hält bis heute an.

Kurz nachdem ich den Podcast gehört hatte, installierte ich die Xamarin-Werkzeuge auf meinem Rechner und legte mit den ersten Experimenten los. Da ich keinen Mac hatte, konnte ich am Anfang nur für Android programmieren. Da mir das nicht genug war, kaufte ich mir einige Monate später einen günstigen Mac und ein günstiges iPhone, um auch für iOS entwickeln zu können.

Ab diesem Zeitpunkt wuchs mein Wunsch, mein Wissen über diese großartige Technologie zu teilen. Ich startete damit, Schulungen und Workshops zum Thema Xamarin zu halten, Fachartikel darüber zu schreiben, auf Konferenzen und User Group Meetings darüber zu sprechen und Videokurse für LinkedIn Learning aufzuzeichnen und Kunden bei der Umsetzung ihrer Projekte zu unterstützen.

Mein Enthusiasmus für das Thema Xamarin und insbesondere das Thema Xamarin.Forms ist seit 2013 ungebrochen. Noch immer freue ich mich darüber, wenn ich Entwicklern etwas über Xamarin beibringen und mein Wissen somit weitergeben kann.

Von der Idee zum Buch

Ein Buch zu schreiben ist harte Arbeit. Härter als ich es erwartet hatte, als ich im Sommer 2016 die Anfrage des Carl Hanser Verlags erhielt, ob ich ein Buch über Xamarin schreiben möchte. Zunächst zögerte ich, denn ich konnte zu diesem Zeitpunkt bereits auf zwei gescheiterte Buchprojekte zurückblicken, die beide nach 130 Seiten abgebrochen wurden. Am Ende siegte jedoch meine Leidenschaft für das Thema Xamarin und nach Rücksprache mit meiner Familie und meiner Lektorin sagte ich zu.

Vom Gedanken, ein Buch zu schreiben, bis zum fertigten Buch dauerte es fast fünf Jahre. Als Außenstehender mag man da vielleicht den Kopf schütteln und sich fragen, warum es so lange dauerte. Natürlich könnte ich Ihnen jetzt erzählen, dass gute Texte ähnlich wie guter Wein eine lange Zeit reifen müssen, aber das wäre einfach Unsinn.

Die wirklichen Ursachen sind eine Kombination verschiedener Gründe. Der Hauptgrund war sicherlich, dass ich noch nie ein Buch (zu Ende) geschrieben hatte und schlichtweg nicht wusste, wie man das macht, ein Buch schreiben. Als Ergebnis machte ich das, was in der Entwicklung von Individualsoftware normalerweise der Auftraggeber macht: Ich wollte einfach zu viel. Meine Vision war es, ein Buch zu schreiben, das Xamarin.Android, Xamarin.iOS, Xamarin.Forms und die Entwicklung von UWP Apps von der Idee bis zum Deployment behandelt.

Die ersten 3 ½ Jahre hielt ich an diesem Plan fest und scheiterte auf voller Linie damit. Der mangelnde Fokus kombiniert mit der Tatsache, dass es eigentlich vier Bücher in einem gewesen wären, der Situation, dass dies mein erstes Buch war und dass ständig, wenn ich ein Kapitel fertig hatte, sich etwas Grundlegendes seitens Microsoft, Apple oder Google änderte, führte dazu, dass ich große Teile des Buches oder der Beispielcodes immer wieder neu schrieb und somit auf der Stelle trat.

Im Frühjahr 2020, als ich das Buch schon fast aufgeben wollte, einigte ich mich mit meiner Lektorin beim Carl Hanser Verlag darauf, das Thema des Buches etwas enger zu fassen und mich auf die Entwicklung von Xamarin.Forms-Apps zu konzentrieren. In diesem Bereich liegt sicherlich auch das Hauptinteresse der Leser, zumindest wenn meine Schulungs- und Beratungsanfragen sowie die Abrufe meiner Videokurse repräsentativ sind. Seit längerer Zeit steigt hier das Interesse an Xamarin.Forms, während Xamarin.Android und Xamarin.iOS so gut wie gar nicht mehr nachgefragt werden.

Als Microsoft auf der Build-Konferenz im Mai 2020 dann .NET MAUI als Evolution von Xamarin.Forms ankündigte und kaum ein Wort über Xamarin.Android oder Xamarin.iOS verlor, war die Strategie klar: Dieses Buch muss sich auf Xamarin.Forms konzentrieren. Also startete ich im Mai 2020 von vorne mit dem Buch und schrieb große Teile der bis dahin 130 geschriebenen Seiten neu.

Mangelnder Fokus und eine zu große Vision waren aber nur eine Ursache für den langen Weg bis zum fertigen Buch. Eine andere war, dass das Schreiben dieses Buches nicht meine Haupttätigkeit war. Stattdessen verdiente ich mein Geld in der Anfangsphase hauptsächlich als Trainer und Berater, ehe ich 2018 die Quality Bytes GmbH, eine Softwarefirma spezialisiert auf die Entwicklung von Apps und Cloud Solutions, gründete und aufbaute, was aufwendiger war als ich dachte. Und das Leben besteht nicht nur aus Arbeit, sondern auch aus Familie. Meine wundervolle Frau Ana und unsere drei großartigen Kinder wollten natürlich auch Aufmerksamkeit von mir und Zeit mit mir verbringen. Das Endergebnis war, dass das Schreiben eine von vielen Tätigkeiten war. Im Laufe der Jahre experimentierte ich mit ver-

schiedensten Techniken, um den Schreibprozess zu optimieren und somit schneller fertig zu werden. Der Durchbruch kam in den letzten zwei Monaten. Nachdem ich nach neun Monaten wieder nur 130 Seiten fertig hatte, schrieb ich den Rest des Buches in zwei Monaten. Die angewandte Optimierungstechnik lässt sich übrigens sehr einfach zusammenfassen: „Einfach machen".

Falls also einer der Leser selbst mit dem Gedanken spielt, sein erstes Buch zu schreiben, dann kann ich ihn oder sie nur herzlich dazu motivieren und folgende Tipps mit auf den Weg geben:

Es ist einfacher, den Umfang des Buches klein zu planen. Nach meiner Erfahrung wächst der Umfang von ganz allein während des Schreibprozesses. Außerdem hilft es ungemein, Gewohnheiten zu schaffen und zum Beispiel jeden Tag ein wenig zu schreiben als sich z. B. vorzunehmen in einer Woche das halbe Buch fertigzustellen.

Danksagung

Wie Sie gerade erfahren haben, war der Weg zu diesem Buch etwas „holprig". Vermutlich wäre es nie erschienen, wenn es nicht zahlreiche Menschen gegeben hätte, die mir geholfen haben und denen ich meinen Dank aussprechen möchte.

Mein größter Dank gilt meiner Frau Ana, die mich während der ganzen Zeit unterstützt, ermutigt und an mich geglaubt hat. Während der letzten fünf Jahre musste sie nicht nur wegen der Arbeit an diesem Buch, sondern auch wegen meiner anderen beruflichen Verpflichtungen häufig auf meine Gesellschaft verzichten. Sie war dabei stets verständnisvoll und spornte mich an, wenn meine Motivation sank. Dass eine Frau ihrem Mann so viele Freiräume für seine persönlichen Projekte einräumt, ist nicht selbstverständlich und ich bin froh, eine so wundervolle Partnerin gefunden zu haben.

Bedanken möchte ich mich auch bei meinen Kindern Joel und Lyandra und bei meinem Stiefsohn Raúl. Ihr wart stets geduldig und verständnisvoll, wenn ich keine Zeit für euch hatte, da ich an diesem Buch arbeitete.

Mein Dank gilt außerdem meiner Mutter Sylvia, die mir im Grundschulalter einen Atari 800XL zusammen mit dem Buch „Spielend Programmieren lernen" kaufte und damit die Grundlage für eine erfolgreiche IT-Berufslaufbahn geschaffen hat. Außerdem las sie jedes Kapitel meiner fehlgeschlagenen Buchprojekte sowie meiner Diplom- und Masterarbeit Korrektur und gab mir wertvolle Tipps, um meinen Schreibstil zu verbessern.

Ein besonderer Dank gilt auch meiner Lektorin Sylvia Hasselbach beim Carl Hanser Verlag, die trotz meiner vorherigen missglückten Buchversuche das notwendige Vertrauen in mich hatte und dieses Buchprojekt mit mir startete. Besonders möchte ich ihre Geduld bei den vielen E-Mails und Telefonaten hervorheben, bei denen ich neue Terminverschiebungen ankündigte. Weiter möchte ich mich bei Walter Saumweber bedanken, der dieses Buch Korrektur las und mir in geduldigen Telefonaten freundlich und gut gelaunt wertvolle Tipps zu meinen Texten gab. Darüber hinaus möchte ich mich auch bei Kristin Rothe und Irene Weilhart vom Carl Hanser Verlag für die gute Zusammenarbeit bedanken.

Außerdem möchte ich meinem Informatiklehrer Gerd Larscheid danken, der mein Interesse an der Programmierung Mitte der 90er Jahre, als ich Schüler in seiner Klasse war, förderte. Er ist der beste Informatiklehrer, den ich jemals kennengelernt habe. Von ihm lernte ich nicht nur programmieren, sondern auch, dass es beim Programmieren nicht nur um das

Schreiben von Quellcode, sondern um das Lösen von (Geschäfts-)Problemen mit der Hilfe von Software geht. Diese Erkenntnis war für meine berufliche Laufbahn unverzichtbar.

Bedanken möchte ich mich auch bei meinem guten Freund Oliver, der dieses Buch direkt nach der Ankündigung vorbestellte und mich in den letzten Jahren immer wieder motivierte, es zu Ende zu schreiben.

Selbstverständlich gibt es nicht nur Menschen im persönlichen Umfeld, denen mein Dank gilt, sondern auch Wegbegleiter aus der Entwickler-Community.

Hier gibt es so viele – aktuelle und ehemalige Kollegen, Sprecher und Aussteller auf Konferenzen –, dass es schwerfällt, jemanden besonders hervorzuheben. Trotzdem möchte ich nicht nur all denjenigen, die zur vorherigen Gruppe gehören, meinen pauschalen Dank aussprechen, sondern einige namentlich nennen. Zum einen wäre da Thomas Claudius Huber, mit dem ich mir während des Microsoft MVP Summits 2014, als ich an meinem Windows-8-Buch schrieb und daran verzweifelte, ein Zimmer teilte. Thomas erzählte mir seine inspirierende Geschichte darüber, wie er sein dreimal so dickes WPF-Buch geschrieben hatte. Seine Erzählungen waren so motivierend, dass ich auch heute noch häufig daran denke.

Weiter wäre da Manfred Steyer, den ich seit vielen Jahren von verschiedenen Konferenzen kenne und der ein erfahrener Autor ist. 2019 hielt er in meiner Firma eine Angular-Schulung. Während des Abendessens an einem der Schulungstage fragte ich ihn nach Tipps und er sagte mir, dass ich am besten erst einmal die Beispiele fertig programmiere. Während der Entwicklung der Beispiele würde mir klarwerden, was ich detailliert beschreiben muss, und was nicht. Mit diesem Hinweis hatte er eindeutig recht und es war dieser Hinweis, dessen Umsetzung mir in den letzten zwei Monaten dabei half, Fahrt aufzunehmen.

Mein Dank gilt weiterhin dem Xamarin-Team für das herausragende Produkt sowie den vielen Freiwilligen in der Community, die auf der ganzen Welt ihr Wissen in der Form von Blog-Beiträgen und Vorträgen teilen, oder ihre Freizeit in Open-Source-Projekte stecken.

Außerdem möchte ich auch Dr. Holger Schwichtenberg für die großartige Zusammenarbeit in den letzten Jahren danken. Er war es, der mich als Autor beim Carl Hanser Verlag ins Gespräch brachte und ohne den es dieses Buch wohl niemals geben würde.

Schlussendlich gilt mein Dank natürlich auch **Ihnen!** Vielen Dank, dass Sie sich für dieses Buch entschieden haben. Wenn Sie zu denen gehören, die mehr oder weniger geduldig auf das Erscheinen dieses Buches gewartet haben, dann möchte ich mich an dieser Stelle bei Ihnen für die lange Wartezeit entschuldigen und hoffe, dass der Inhalt die Wartezeit wieder wettmacht.

André Krämer

Mai 2021

Der Autor

André Krämer startete seine berufliche Laufbahn 1997 mit einer Ausbildung zum Fachinformatiker für Anwendungsentwicklung, nachdem er zuvor auf der höheren Berufsfachschule für Datenverarbeitung den Abschluss als staatlich geprüfter kaufmännischer Assistent für Datenverarbeitung machte.

Nach seiner Ausbildung bei einem großen IT-Systemhaus arbeitete er zunächst als Entwickler und später als technischer Teamleiter in seinem Ausbildungsbetrieb. Bereits damals fokussierte er sich auf Microsoft-Technologien.

2002 wechselte er zu einem international tätigen Softwarehaus für Software im industriellen Qualitätsmanagement und sammelte dort umfangreiche Erfahrungen mit Microsoft .NET ab der Version 1.0. Während dieser Zeit studierte er nebenberuflich an der FH Köln Wirtschaftsinformatik und erlangte seinen Abschluss als Diplom-Informatiker (FH).

2008 bis 2011 arbeitete er für ein internationales Beratungsunternehmen als Senior Application Architect und absolvierte währenddessen ein nebenberufliches Studium zum Master of Science Wirtschaftsinformatik an der TH Kön.

2012 machte er sich selbständig, wurde Partner im *www.IT-Visions.de*-Expertennetzwerk, und arbeitete bis Sommer 2018 als Trainer und Berater. Während dieser Zeit führte er überwiegend Individualschulungen für Softwareentwickler durch und schulte vor allem die Themen App-Entwicklung mit Xamarin, Web-Entwicklung mit ASP.NET und DevOps mit Azure DevOps. Im Rahmen seiner Tätigkeit veröffentlichte er über 20 Videokurse für LinkedIn Learning, schrieb zahlreiche Fachartikel und sprach regelmäßig auf Entwicklerkonferenzen.

2013 erhielt er für sein Fachwissen und Community-Engagement von Microsoft die Auszeichnung zum Microsoft Most Valuable Professional (MVP), und wurde seitdem jedes Jahr erneut mit dem Titel ausgezeichnet.

QUALITYBYTES

2018 wagte er den nächsten Schritt und gründete die Quality Bytes GmbH, ein Softwarehaus in Bad Breisig am Rhein zwischen Bonn und Koblenz, und übernahm dort die Geschäftsführung. Mit seinem mittlerweile 15-köpfigen Team schreibt er dort Individualsoftware auf Projektbasis für seine Kunden.

Der Fokus der Quality Bytes GmbH liegt auf der Entwicklung von Mobile Apps für Android, iOS und Windows mit Xamarin, der Entwicklung von Webportalen für die Microsoft Azure Cloud oder das Selbsthosting auf der Basis von Angular und ASP.NET Core sowie der Implementierung von SAP-Integrationslösungen.

Wenn Sie Projektunterstützung in einem der genannten Bereiche benötigen, dann freuen sich André Krämer und sein Team über Ihre Nachricht unter *info@qualitybytes.de* oder den Besuch des Internetauftritts der Quality Bytes GmbH, den Sie unter *https://qualitybytes.de* finden.

Sein aktuelles Interesse im Bereich der Softwareentwicklung gilt neben Xamarin, ASP.NET und Azure DevOps den Themen Azure und Docker sowie agilen Entwicklungsmethoden.

Seine Freizeit verbringt André Krämer am liebsten mit seiner Familie. Nachdem die beiden ältesten Kinder bereits erwachsen und ausgezogen sind, beobachten er und seine Frau mit großer Freude die fußballerischen Aktivitäten ihres jüngsten Sohns. Alle drei sind daher häufig auf den Fußballplätzen rund um Koblenz anzutreffen.

11 Bilder, Icons und Schriftarten

Bilder sind ohne Zweifel ein wichtiger Bestandteil vieler Apps. Mit Ihrer Hilfe kann die eigene Marke in der App durch die Anzeige zum Beispiel eines Logos durchgesetzt werden, Bilder können die Aufmerksamkeit der Anwender erregen und diese dadurch zur Nutzung der App animieren, sie können in der Form von Fotos als Daten der App dienen oder sie einfach nur dekorieren.

Sie sehen also, das Einsatzgebiet von Bildern ist in mobilen Apps sehr umfangreich. Ein weitere Alternative zum Aufwerten einer App sind eigene Schriftarten. Über ihren Einsatz können Sie sich mit Ihren Apps von anderen Apps abheben, die nur Standardschriftarten nutzen.

Ähnlich wie bei Bildern ist der Einsatzbereich auch bei Schriftarten recht umfangreich. Sie können in Ihrer App durch Schriftarten zum Beispiel Ihre Marke unterstreichen, indem Sie Schriftarten aus Ihrem Corporate Design einsetzen, Sie können außergewöhnliche Schriftarten nutzen, um Ihrer App mit einem besonderen Design zu versehen, oder Icons mithilfe von Schriftarten darstellen.

11.1 Bilder darstellen

Bilder werden in Xamarin.Forms-Apps mithilfe des Image-Steuerelements dargestellt. Die Quelle des Bildes wird über die Eigenschaft `Source` vom Typ `ImageSource` festgelegt. Eine Quelle kann entweder eine Datei, die in einem der plattformspezifischen Projekte liegt, eine Webadresse, eine eingebettete Ressource oder ein Stream sein.

11.1.1 Plattformspezifische Bilder

Die beste, allerdings auch aufwendigste Darstellung erhalten Sie über plattformspezifische Bilder. Bei dieser Variante erreichen Sie die besten Ergebnisse, da Sie verschiedene Bilder für unterschiedliche Auflösungen und Pixeldichten der Geräte bereitstellen können. Dadurch können Sie sicherstellen, dass auf jedem Endgerät ein optimales Ergebnis erzielt wird.

Sie müssen im Xamarin.Forms-Markup nicht darauf eingehen, dass Sie unterschiedliche Bilder in unterschiedlichen Auflösungen vorliegen haben. Sie geben lediglich über die Eigenschaft `Source` den Namen der Bilddatei an. Den Rest erledigen die verschiedenen Betriebssystem-APIs beim Laden der Bilddatei automatisch.

```
<StackLayout >
  <Image Source="qualitybytes_sample.png" VerticalOptions="CenterAndExpand"></Image>
</StackLayout>
```

Die aus dem Code resultierende Ausgabe des Bildes sehen Sie in Bild 11.1.

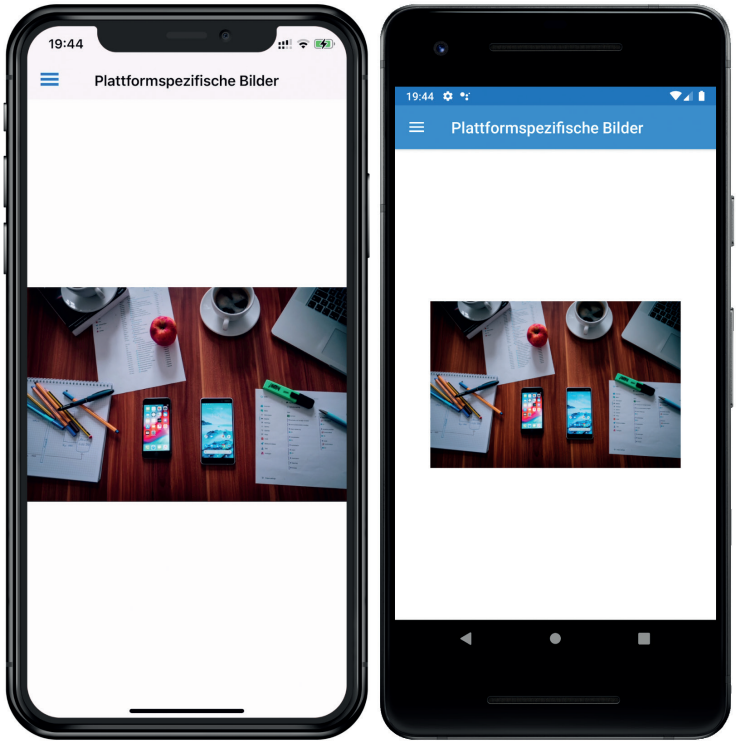

Bild 11.1 Ein Bild in einer Xamarin.Forms-App

Nachdem wir nun gesehen haben, was wir im plattformspezifischen Projekt machen müssen, um ein Bild anzuzeigen, sehen wir uns die notwendigen Schritte unter Android und iOS an.

Android

Unter Android legen Sie Ihre Bilder in Unterordnern des Ordners *Resources\drawable* ab. Je unterstützter Pixeldichte erstellen Sie einen Ordner nach der Namensgebung aus Tabelle 11.1.

Tabelle 11.1 Pixeldichten und zugehörige Ordner für Android-Grafiken

Ordner	Beschreibung	Pixeldichte	Faktor	Beispiel
ldpi	low-density (geringe Pixeldichte)	120 dpi	0,75	225 x 150 Pixel
mdpi	medium-density (mittlere Pixeldichte; Basiswert)	160 dpi	1,0	300 x 200 Pixel
hdpi	high-density (hohe Pixeldichte)	240 dpi	1,5	450 x 300 Pixel
xhdpi	extra-high-density (extra hohe Pixeldichte)	320 dpi	2,0	600 x 400 Pixel
xxhdpi	extra-extra-high-density (extra extra hohe Pixeldichte)	480 dpi	3,0	900 x 600 Pixel
xxxhdpi	extra-extra-extra-high-density (extra extra extra hohe Pixeldichte)	640 dpi	4	1200 x 800 Pixel
nodpi	Ressourcen, die unabhängig von der Pixeldichte immer gleich dargestellt werden sollen			

Innerhalb der Ordner legen Sie Ihre Dateien ab. Wenn Sie eine Datei in mehreren Pixeldichten vorhalten möchten, dann muss diese unter demselben Namen in unterschiedlichen Auflösungen in die jeweiligen Ordner kopiert werden. Außerdem muss in den Dateieigenschaften der Eigenschaft *Buildvorgang* der Wert `AndroidResource` zugewiesen werden. Sie müssen nicht zwangsläufig für jede Pixeldichte auch ein Bild vorhalten. Bei fehlenden Bildern wird automatisch das mit der passendsten Pixeldichte geladen und entsprechend skaliert. Da Visual Studio bei der Projektanlage nur den Ordner *Resources\drawable* anlegt, müssen Sie die verschiedenen Unterordner selbst anlegen. Dabei müssen Sie zwingend auf die korrekte Schreibweise der Ordnernamen achten.

Bild 11.2 zeigt eine entsprechende Umsetzung. In diesem Beispiel wurde das Bild *qualitybytes_sample.png* in einer Auflösung von 1280 × 854 Pixeln in den Ordner *xxxhdpi* kopiert. Anschließend wurde es entsprechend des Faktors aus Tabelle 11.1 auf die kleineren Auflösungen skaliert und in die entsprechenden Ordner kopiert.

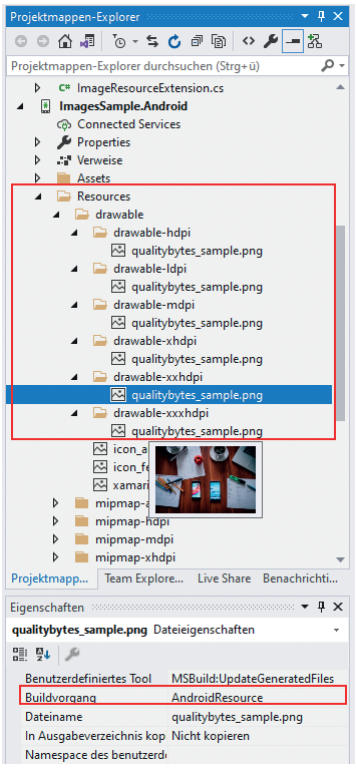

Bild 11.2
Der Ordner „drawable“ mit seinen Unterordnern

Dateinamen für Bilder unter Android

Android unterstützt bei den Dateinamen nur Kleinbuchstaben, Zahlen, Punkte und den Unterstrich. Andere Zeichen werden nicht unterstützt. Um die Cross-Plattform-Kompatibilität zu erhalten, sollten Sie die gleichen Regeln auch für Ihre Bilder unter iOS anwenden.

iOS

Unter iOS benötigen Sie im Gegensatz zu Android nur drei unterschiedliche Pixeldichten für Bilder: die einfache, die doppelte und die dreifache. Standardmäßig benennen Sie ihre Bilder für iOS nach dem folgenden Schema:

- *<Name>.<Endung>*, z. B. *qualitybytes_sample.png*
- *<Name>**@2x**.<Endung>*, z. B. *qualitybytes_sample**@2x**.png*
- *<Name>**@3x**.<Endung>*, z. B. *qualitybytes_sample**@3x**.png*

Die Bilder legen Sie anschließend in einem Ressourcenkatalog vom Typ *Bildergruppe* (engl. *image set*) ab. Zu diesem Zweck öffnen Sie in Ihrem iOS-Startprojekt den bestehenden Ressourcenkatalog mit dem Namen *Assets*, oder Sie legen einen neuen Ressourcenkatalog

(*.xcassets*-Datei) an. Innerhalb des Ressourcenkatalogs erzeugen Sie einen neuen Eintrag vom Typ *Bildergruppe* und geben ihm den Namen Ihrer Bilderdatei (ohne Endung), also z. B. *qualitybytes_sample*. Anschließend öffnet sich ein Bereich mit leeren Kästen für die drei Auflösungen Ihres Bildes. In diese Kästchen können Sie per Drag-and-drop einfach die passenden Bilder ziehen.

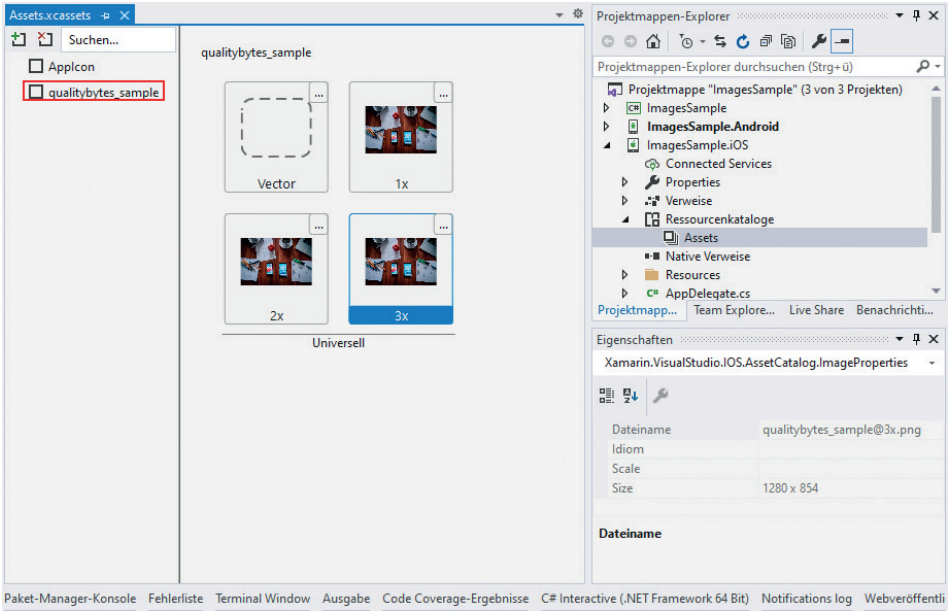

Bild 11.3 Bilder unter iOS

iOS-Bildergruppen und Xamarin Hot Restart

In der zum Zeitpunkt der Drucklegung dieses Buches aktuellen Visual-Studio-Version gibt es eine Einschränkung bei der Kombination von Bildergruppen und Xamarin Hot Restart. Während des Debuggens mit einem an einen Windows-Rechner angeschlossenen iOS-Gerät werden keine Bilder angezeigt.

Dieses Problem besteht nur währen des Debuggings und nicht in der ausgelieferten App. Wenn Sie während des Debuggings trotzdem Bilder sehen möchten, dann müssen Sie auf den Einsatz von Hot Restart verzichten und Ihr Mobilgerät an einem Mac anschließen, auf dem Sie die App (remote) kompilieren und debuggen.

11.1.2 Eingebettete Bilder

Der im vorherigen Abschnitt erläuterte Weg über plattformspezifische Bilder ist, wie bereits erwähnt, der beste, aber auch der aufwendigste Weg. Deutlich weniger Aufwand verursachen eingebettete Bilder. Bei einem eingebetteten Bild wird dieses nur in einer Auflösung, idealerweise der höchsten, vorgehalten und als eingebettete Ressource in das plattformübergreifende Xamarin.Forms-Projekt eingefügt. Beim Laden wird es automatisch auf die passende Größe skaliert.

Eingebettete Bilder werden in die Assembly hineinkompiliert. Dies geschieht, wenn Sie die Eigenschaft *Buildvorgang* innerhalb der Dateieigenschaften auf den Wert `Eingebettete Ressource` setzen.

Bild 11.4 zeigt, wie dies in der Praxis aussieht.

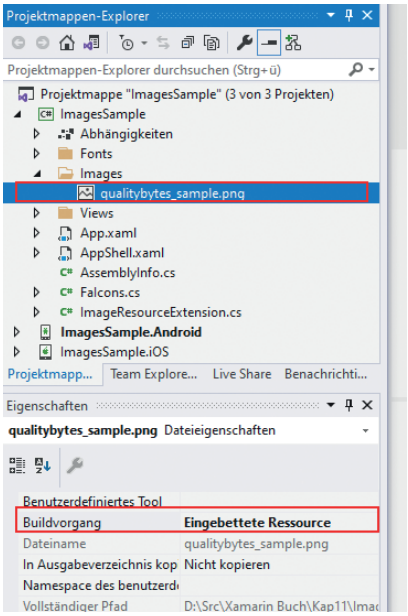

Bild 11.4
Ein eingebettetes Bild

Leider können Sie eingebettete Bilder nicht direkt in XAML verwenden, da es keine eingebaute Konvertierung eines Strings zu einer `ResourceImageSource` gibt. Aus diesem Grund müssen Sie Ihre Bilder entweder im Code-Behind laden oder eine XAML-Markuperweiterung schreiben.

Eine Markuperweiterung erlaubt es Ihnen, Werte von Attributen in Ihrem Quellcode imperativ zu erzeugen, statt deklarativ zu beschreiben. Markuperweiterungen sind Klassen, die die Schnittstellen `IMarkupExtension` oder `IMarkupExtension<T>` implementieren. Beide Schnittstellen definieren die Methode `ProvideValue`, deren Aufgabe es ist, den Wert für die aufrufende XAML-Eigenschaft zurückzugeben.

Für die Anzeige von eingebetteten Bildern gibt es in der offiziellen Dokumentation bereits eine bespielhafte Implementierung einer Markuperweiterung, die Sie in leicht abgewandelter Form in Listing 11.1 sehen.

Listing 11.1 Mit der ImageResourceExtension-Markuperweiterung können Sie eingebettete Bilder im XAML laden.

```
[ContentProperty(nameof(Source))]
public class ImageResourceExtension : IMarkupExtension
{
  public string Source { get; set; }

  public object ProvideValue(IServiceProvider serviceProvider)
  {
    if (Source == null)
    {
      return null;
    }

    return ImageSource.FromResource(Source,
      typeof(ImageResourceExtension).GetTypeInfo().Assembly);
  }
}
```

Die Anwendung der Markuperweiterung aus Listing 11.1 sehen Sie in Listing 11.2. Wie Sie im Listing sehen, müssen Sie zunächst einen Alias für den vollqualifizierten Namensraum der Markuperweiterung angeben, im Beispiel ist dies `local`.

Anschließend können Sie die Markuperweiterung über Datenbindung nutzen. Als Argument übergeben Sie den vollqualifizierten Namen des eingebetteten Bildes in der Form `<AssemblyName>.<OrdnerName>.<BildName>.<Endung>`, also zum Beispiel `ImagesSample.Images.qualitybytes_sample.png`.

Listing 11.2 ImageResource-Markuperweiterung

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:local="clr-namespace:ImagesSample;assembly=ImagesSample"
       x:Class="ImagesSample.Views.EmbeddedImagesSamplePage"
       Title="Eingebettete Bilder">
  <ContentPage.Content>
    <StackLayout>
      <Image
        Source="{local:ImageResource ImagesSample.Images.qualitybytes_sample.png}"
        VerticalOptions="CenterAndExpand" />
    </StackLayout>
  </ContentPage.Content>
</ContentPage>
```

Dynamische Anzeige über ValueConverter

Wenn Sie die Quelle für Ihr Bild nicht statisch, sondern dynamisch per Datenbindung übergeben möchten, dann müssen Sie dazu einen Wertkonvertierer (ValueConverter) implementieren. Dabei handelt es sich um eine Klasse, die die Schnittstelle `IValueConverter` implementiert. Die Schnittstelle definiert die beiden Methoden `Convert` und `ConvertBack`. Die Methode `Convert` konvertiert den gebundenen Wert in eine entsprechende XAML-Struktur, die Methode `ConvertBack` macht das Gegenteil.

Listing 11.3 zeigt eine sehr einfache Implementierung eines ValueConverters für eine ImageSource. Der vorliegende Beispielcode geht davon aus, dass ein gültiger Bildpfad per Datenbindung übergeben wird.

Listing 11.3 Ein einfacher ValueConverter für eine ImageSource

```
public class SimpleResourceImageSourceConverter : IValueConverter
{

  public object Convert(object value, Type targetType, object parameter,
    CultureInfo culture)
  {
    return ImageSource.FromResource(value,
         typeof(ResourceImageSourceConverter).GetTypeInfo().Assembly);
  }

  public object ConvertBack(object value, Type targetType, object parameter,
    CultureInfo culture)
  {
    throw new NotImplementedException();
  }
}
```

Der Einsatz des ValueConverters im XAML-Code würde wie folgt aussehen – ausgehend davon, dass es im ViewModel eine Eigenschaft `ImagePath` gibt:

```
<Image Source="{Binding ImagePath}"/>
```

■ 11.2 Schriftarten

Ohne weitere Konfiguration werden alle Texte, die Sie in Ihrer App ausgeben, mit der Systemschriftart des jeweiligen mobilen Betriebssystems dargestellt. Durch die Nutzung der Systemschriftart wird sich Ihre App auf der einen Seite zwar einheitlich in das Gesamtökosystem der mobilen Plattform einreihen, auf der anderen Seite wird sie jedoch eintönig wirken.

Über den Einsatz eigener Schriftarten haben Sie die Möglichkeit, sich etwas vom Einheitsbrei abzusetzen und das Corporate Design Ihrer Marke zu unterstreichen, indem Sie zum Beispiel Ihre Hausschriftart nutzen.

Vorsicht bei übermäßiger Verwendung benutzerdefinierter Schriftarten

Wie bei so vielen Dingen sollte auch der Einsatz benutzerdefinierter Schriftarten in Maßen erfolgen. Für den Anwender Ihrer Apps kann es anstrengend sein, wenn Sie die Schriftarten zu häufig wechseln oder Ihre komplette App in zum Beispiel *Comic Sans MS* darstellen. Im schlimmsten Fall erreichen Sie durch den übermäßigen Einsatz sogar eine abschreckende Wirkung.

Daher sollten Sie gut überlegen, an welchen Stellen der Einsatz einer eigenen Schriftart Sinn macht. Auffallende Schriftarten werden häufig für Überschriften genutzt, für Fließtext kommen jedoch meist schlichte Schriftarten zum Einsatz. Ziehen Sie im Zweifelsfall einen Designer zu Rate.

Die zu verwende Schriftart definieren Sie für sämtliche Steuerelemente, die Text darstellen können, über die Eigenschaft `FontFamily`. Da nicht jede Schriftart auf allen mobilen Endgeräten vorinstalliert ist, ist es bei benutzerdefinierten Schriftarten meist Erforderlich, dass Sie die Schriftart mit Ihrer App ausliefern.

Dazu sind zwei Schritte notwendig. Als Erstes müssen Sie die Schriftart als *.ttf-* oder *.otf-*Datei zu Ihrem plattformübergreifend geteilten Projekt hinzufügen und in den Dateieigenschaften der Eigenschaft *Buildvorgang* den Wert `Eingebettete Ressource` zuweisen (siehe Bild 11.5). In welchem Ordner Sie Die Datei ablegen, ist Ihnen überlassen. In der Praxis wird häufig ein Ordner *Fonts* oder die Ordnerstruktur *Resources\Fonts* angelegt und für eigene Schriftarten genutzt.

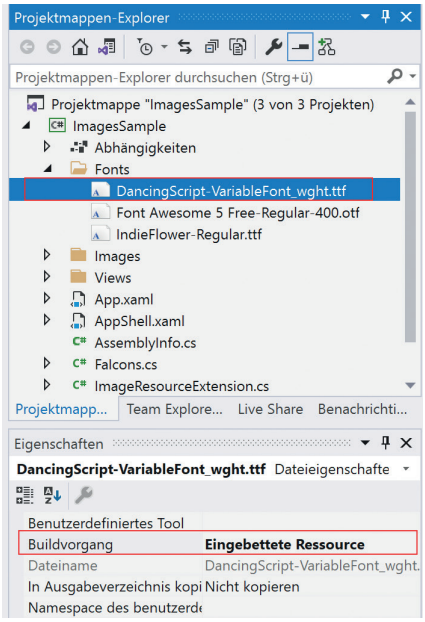

Bild 11.5
Benutzerdefinierte Schriftarten müssen als eingebettete Ressource zum Projekt hinzugefügt werden.

Nachdem die Schriftart zum Projekt hinzugefügt wurde, müssen Sie sie im nächsten Schritt registrieren. Dazu öffnen Sie die Datei *AssemblyInfo.cs* Ihres geteilten Projekts und fügen dort je Schriftart das Attribut `ExportFont` hinzu.

Dem Attribut übergeben Sie den Namen der Schriftartendatei sowie einen optionalen Alias, über den Sie die Schriftart im Projekt referenzieren. Die genaue Syntax sehen Sie in Listing 11.4.

Listing 11.4 Registrierung eigener Schriftarten in der Datei AssemblyInfo.cs

```
using Xamarin.Forms;
[assembly: ExportFont("DancingScript-VariableFont_wght.ttf", Alias =
"DancingScript")]
[assembly: ExportFont("Font Awesome 5 Free-Regular-400.otf", Alias = "FontAwesome")]
[assembly: ExportFont("IndieFlower-Regular.ttf", Alias = "IndieFlower")]
```

Nachdem beide Schritte erfolgt sind, können Sie die registrierten Schriftarten in Ihren Bildschirmmasken verwenden. Dazu müssen Sie lediglich die Eigenschaft `FontFamily` für Ihre Steuerelemente angeben und ihr als Wert den zuvor registrierten Alias Ihrer Schriftarten zuweisen.

```
<Label Text="Hallo Xamarin.Forms (Dancing Script)!" FontFamily="DancingScript" />
<Label Text="Hallo Xamarin.Forms (Indie Flower)!" FontFamily="IndieFlower" />
```

Bild 11.6 zeigt den Einsatz der beiden zuvor definierten Schriftarten.

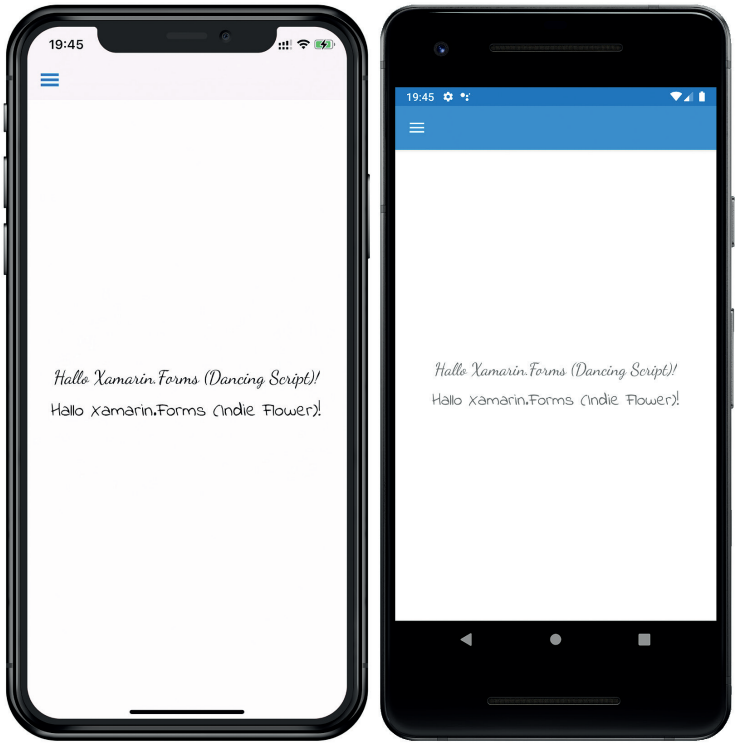

Bild 11.6 Benutzerdefinierte Schriftarten in einer Xamarin.Forms-App

Quellen für benutzerdefinierte Schriftarten und Lizenzen

Bevor Sie eine benutzerdefinierte Schriftart in Ihren Projekten einsetzen, sollten Sie überprüfen, ob Sie auch die Lizenz für den Einsatzzweck in einer

mobilen Anwendung haben. Sie können zum Beispiel nicht einfach jede beliebige Schriftart aus Ihrem Windows-Betriebssystem als *.ttf-* oder *.otf*-Datei exportieren und anschließend in Ihren Apps nutzen. Der Einsatzbereich der Schriftarten ist nämlich häufig auf die Nutzung unter Windows eingeschränkt.

Als Alternative gibt es glücklicherweise viele Webseiten im Internet, auf denen Sie Schriftarten für die Nutzung in mobilen Apps kostenpflichtig lizensieren können. Darüber hinaus gibt es sogar einige Webseiten, wie zum Beispiel Google Fonts *(https://fonts.google.com),* die Schriftarten zum Download anbieten, die Sie kostenfrei in Ihren Apps nutzen dürfen.

11.3 Icon-Schriftarten verwenden

Ein Spezialfall benutzerdefinierter Schriftarten ist der Einsatz von Icon-Schriftarten. Über Icon-Schriftarten haben Sie die Möglichkeit, Symbole in Ihrer App darzustellen, so wie Sie es in Bild 11.7 sehen.

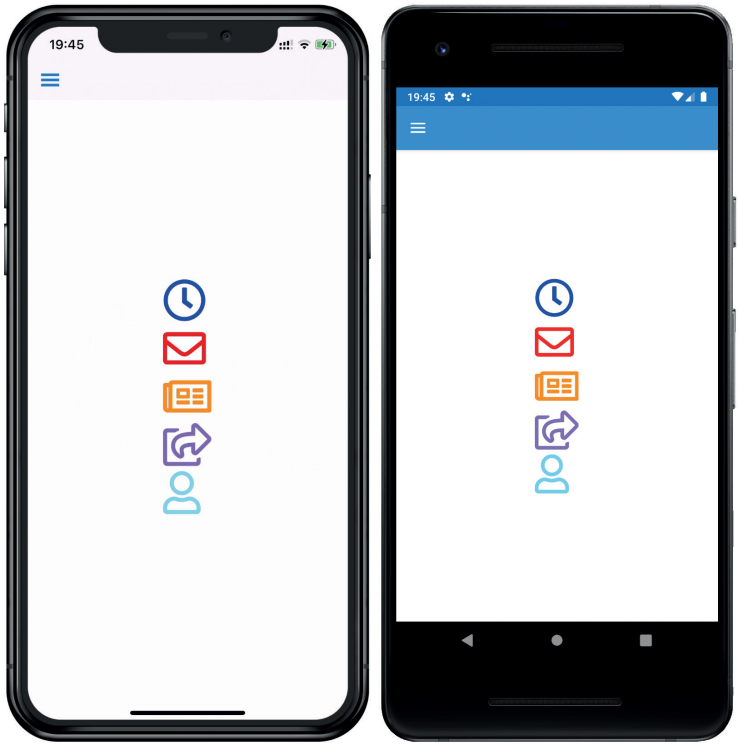

Bild 11.7 Über Icon-Schriftarten können Sie Symbole in Ihren Apps darstellen.

Ursprünglich wurden Symbole als Grafiken zu Apps hinzugefügt und über ein Image-Element dargestellt. Das Problem an dieser Vorgehensweise ist, dass Sie Grafiken mehrfach vorhalten müssen, wenn Sie die Symbole in unterschiedlichen Farben oder Größen darstellen möchten. Soll ein Symbol, wenn es aktiv ist, zum Beispiel in Schwarz dargestellt werden und wenn es inaktiv ist in Grau, dann müssen Sie die Grafik schon doppelt vorhalten. Um eine möglichst gute Darstellung auf unterschiedlichen Bildschirmauflösungen zu gewährleisten, müssen Sie beide Grafiken je sechsmal unter Android und dreimal unter iOS ablegen, womit wir bereits bei 18 Dateien für ein Symbol wären.

Diese Herausforderung lässt sich glücklicherweise mit Icon-Schriftarten ohne viel Aufwand lösen. Icon-Schriftarten sind Schriftarten, die anstatt oder zusätzlich zu Buchstaben auch Symbole definieren. Bekannte Beispiele hierfür sind die Google Icon Fonts *(https://fonts.google.com/icons)* oder Font Awesome *(https://fontawesome.com)*. Eine grobe Vorstellung über die Art von Symbolen, die Sie nutzen können, gibt Ihnen Bild 11.8.

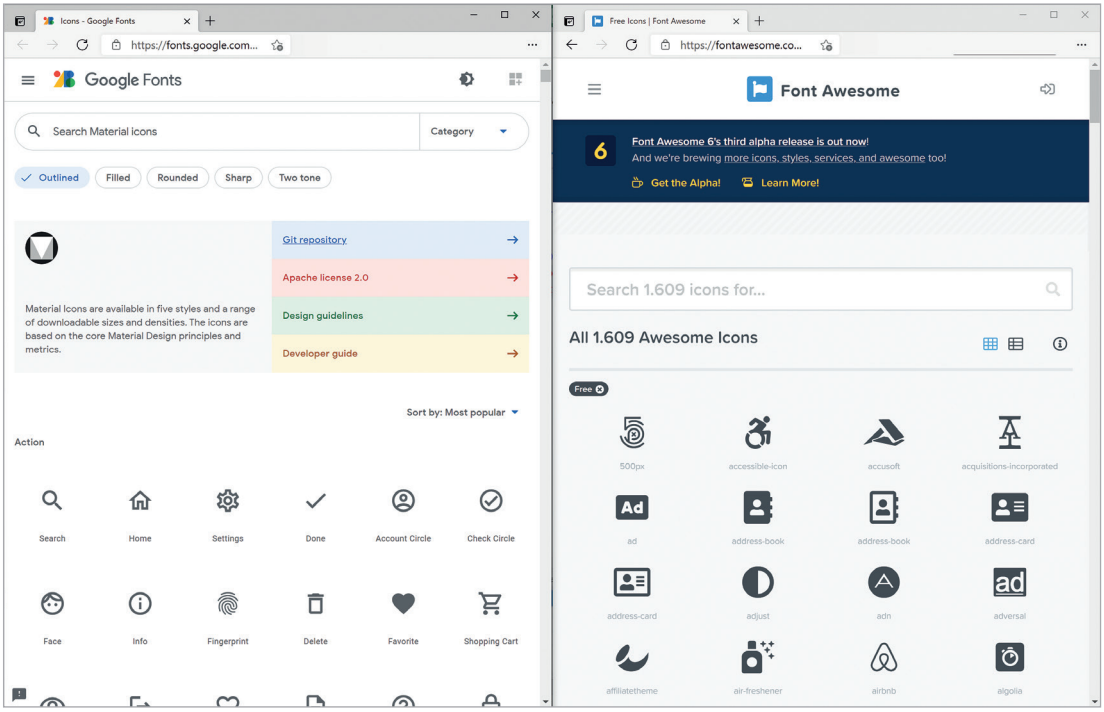

Bild 11.8 Beispiele der Icon-Schriftarten Google Icon Fonts und Font Awesome

Der große Vorteil bei der Nutzung von Icon-Schriftarten liegt darin, dass Sie die Größe des Symbols über die Schriftgröße verlustfrei variieren können. Da es sich bei Schriftarten um Vektoren handelt, können Sie also wahlweise eine sehr kleine oder eine sehr große Darstellung wählen, ohne dass Sie einen Qualitätsverlust sehen.

Darüber hinaus können Sie die Farbe des Symbols über die Schriftfarbe variieren. Listing 11.5 zeigt den Einsatz der Symbolschriftart *Font Awesome* in verschiedenen Farben.

Listing 11.5 Praktischer Einsatz von Icon-Schriftarten

```
<Label FontFamily="FontAwesome" Text="{x:Static local:FaIcons.Clock}"
  FontSize="50" TextColor="Blue"  />

<Label FontFamily="FontAwesome" Text="{x:Static local:FaIcons.Envelope}"
  FontSize="50" TextColor="Red"/>

<Label FontFamily="FontAwesome" Text="{x:Static local:FaIcons.Newspaper}"
  FontSize="50" TextColor="DarkOrange" />

<Label FontFamily="FontAwesome" Text="{x:Static local:FaIcons.ShareSquare}"
  FontSize="50" TextColor="MediumPurple" />

<Label FontFamily="FontAwesome" Text="{x:Static local:FaIcons.User}"
  FontSize="50" TextColor="SkyBlue"  />
```

Wie Sie in Listing 11.5 sehen, unterscheidet sich der Einsatz von Icon-Schriftarten nicht grundlegend vom Einsatz anderer benutzerdefinierter Schriftarten. Auch in diesem Fall müssen Sie die Schriftart als *.ttf*- oder *.otf*-Datei als eingebettete Ressource bereitstellen und anschließend in Ihren Steuerelementen die entsprechende Schriftart angeben. Was auf den ersten Blick ungewohnt aussieht, ist die Zuweisung des Attributs `Text`: `Text="{x:Static local:FaIcons.Clock}"`.

Der Hintergrund für diese Schreibweise ist der folgende: Jedes Symbol hat einen eigenen Unicode-Zeichencode. Diesen können Sie in der Regel auf der Webseite des Anbieters der Schriftart nachlesen. Der Zeichencode für die Uhr ist zum Beispiel `f017` bzw. `\uf017`, da es sich um Unicode handelt. In XAML müssen Sie das Unicode-Escape-Zeichen `\u` wie folgt codieren: `&#x`. Zum Anzeigen der Uhr wäre demnach folgender Code notwendig: `Text="&#xf017"`. Sie stimmen mir sicherlich zu, dass diese Schreibweise nicht sonderlich lesbar ist. Je mehr Symbole Sie in Ihrer App verwenden, desto kryptischer wird Ihr XAML-Code.

Eine Alternative dazu ist der Einsatz einer statischen Klasse, die entsprechende Konstanten definiert. Diese können Sie dann in Ihrem XAML-Code verwenden (Listing 11.6).

Listing 11.6 Eine statische Klasse, die Konstanten für Symbole definiert

```
static class FaIcons
{
  public const string Newspaper = "\uf1ea";
  public const string ShareSquare = "\uf14d";
  public const string Envelope = "\uf0e0";
  public const string User = "\uf007";
  public const string Clock = "\uf017";
}
```

Über die Webseite *https://andreinitescu.github.io/IconFont2Code* können Sie auf sehr komfortable Weise eine solche statische Klasse erzeugen, indem Sie auf der Seite Ihre Icon-Schriftart hochladen und anschießend die gewünschten Icons auswählen. Die Webseite generiert dann automatisch eine entsprechende C#-Klasse (siehe Bild 11.9), die Sie in Ihrem Projekt einbinden können.

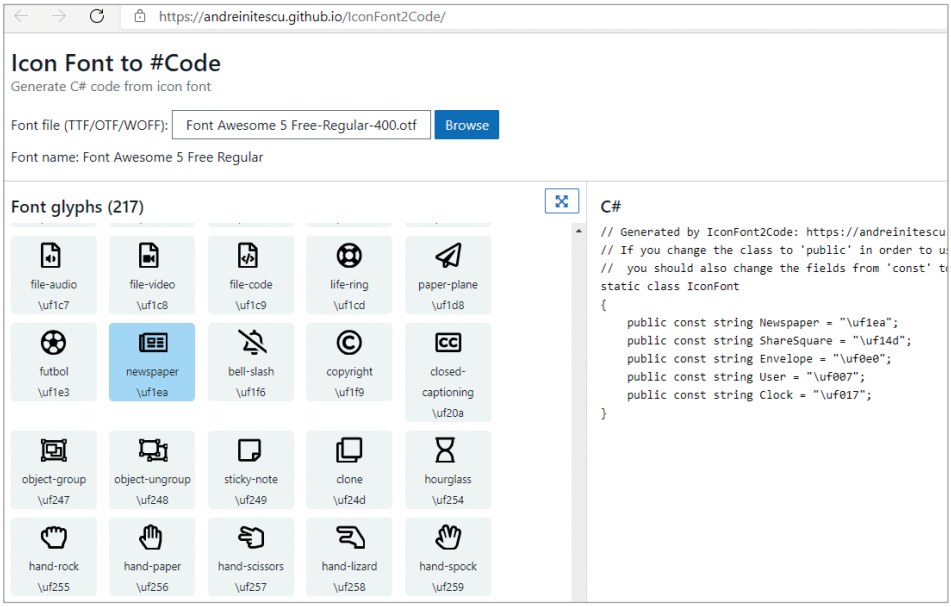

Bild 11.9 Über die Webseite „Icon Font to #Code“ können Sie im Handumdrehen statische Klassen für Ihre Symbolschriftarten erstellen.

In einigen Fällen, zum Beispiel bei den Menüeinträgen der Shell, erwartet das Steuerelement die Angabe einer `ImageSource` für ein Icon. Auf den ersten Blick scheint der Einsatz einer Symbolschriftart an dieser Stelle unmöglich zu sein. Glücklicherweise gibt es jedoch die Klasse `FontImageSource`, mit der Sie ein Icon aus einer Icon-Schriftart an eine `ImageSource`-Eigenschaft übergeben können.

Im Fall eines Shell-Menüeintrags sieht dies wie folgt aus:

```
<ShellContent Title="Profil" ContentTemplate="{DataTemplate profile:ProfilePage}">
  <ShellContent.Icon>
    <FontImageSource FontFamily="Fa-Solid"
      Glyph="{x:Static fonts:FaSolid.UserCircle}" Color="Black"></FontImageSource>
  </ShellContent.Icon>
</ShellContent>
```

Wie Sie sehen, hat die Klasse `FontImageSource` zwei wichtige Eigenschaften. Über die Eigenschaft `FontFamiliy` geben Sie die zu verwendende Symbolschriftart an. Über die Eigenschaft `Glyph` geben Sie das anzuzeigende Symbol an.

11.4 Bilder und Schriftarten in der Beispiel-App

In diesem Kapitel geben wir der Oberfläche der Beispiel-App den letzten Schliff. Wir setzen dazu auf dem fertigen Quellcode aus Kapitel 10, den Sie in den Beispieldateien zu diesem Buch unter *Kap10\ElVegetarianoFurio* finden, auf und kümmern uns zunächst um das Menü und die Startseite.

Vorbereitende Arbeiten

Als vorbereitende Arbeiten kopieren wir den Ordner *Images* und den Ordner *Fonts* aus dem Ordner *Beispieldaten* in unser plattformübergreifend geteiltes Projekt *ElVegetarianoFurio*.

Im Ordner *Images* gibt es die beiden Unterordner *Categories* und *Dishes*. In beiden Ordnern gibt es durchnummerierte Bilddateien. Es handelt sich dabei um die Bilder passend zur ID des jeweiligen Eintrags aus unserem `DummyDataService`. Für jedes der Bilder stellen wir in den Dateieigenschaften den *Buildvorgang* auf den Wert `Eingebettete Ressource`.

Um die Bilder via Datenbindung passend zur jeweiligen Speise oder Kategorie laden zu können, legen wir den `ValueConverter` mit dem Namen `ResourceImageSourceConverter` aus Listing 11.8 an.

Listing 11.7 Die Klasse ResourceImageSourceConverter ermöglicht das Laden von eingebetteten Bildern über Datenbindung.

```
public class ResourceImageSourceConverter : IValueConverter
{
  public object Convert(object value, Type targetType,
         object parameter, CultureInfo culture)
  {
    if (parameter is string source && !string.IsNullOrEmpty(source))
    {
      source = string.Format(source, value);
    }
    else
    {
      source = value.ToString();
    }

    return ImageSource.FromResource(source,
      typeof(ResourceImageSourceConverter).GetTypeInfo().Assembly);
  }

  public object ConvertBack(object value, Type targetType,
                            object parameter, CultureInfo culture)
  {
    throw new NotImplementedException();
  }
}
```

Die Arbeitsweise des Converters ist wie folgt: In der Methode `Convert` wird zunächst geprüft, ob ein Parameter vom Typ `String` übergeben wurde, der nicht leer ist. Falls dem so ist, dann geht der Code davon aus, dass es sich bei dem Wert um einen formatierbaren

String mit einem Platzhalter handelt, zum Beispiel: `"ElVegetarianoFurio.Images.Categories.{0}.jpg"`. Über den Aufruf von `string.Format(source, value);` wird dieser Platzhalter durch den gebundenen Wert, also zum Beispiel der ID einer Kategorie, ersetzt. Somit wird ein vollständiger Pfad zu einer eingebetteten Bilddatei erzeugt. Wird hingegen kein Parameter übergeben, oder ist dieser nicht vom Typ String, oder ein leerer String, dann wird als Quelle der gebundene Wert angenommen.

In der Praxis kann die Nutzung des Converters im XAML-Code wie folgt aussehen:

```
<Image Source="{Binding Id, Converter={StaticResource ResourceImageSourceConverter},
ConverterParameter='ElVegetarianoFurio.Images.Categories.{0}.jpg'}"/>
```

Im vorliegenden Fall wird als Parameter der Wert `"ElVegetarianoFurio.Images.Categories.{0}.jpg"` übergeben und die ID des Datensatz gebunden. In der Methode `Convert` würde dies zum Beispiel zu folgender Bildquelle führen, wenn `Id` den Wert 1 hat: `"ElVegetarianoFurio.Images.Categories.1.jpg"`.

Um den Converter innerhalb des Markups nutzen zu können, müssen wir ihn in der Datei *App.xaml* registrieren. Dazu fügen wir die fett gedruckten Zeilen des nachfolgenden Quellcodes ein:

```
<Application xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
       xmlns:local="clr-namespace:ElVegetarianoFurio;assembly=ElVegetarianoFurio"
       x:Class="ElVegetarianoFurio.App">
  <Application.Resources>
    <ResourceDictionary>
      <local:ResourceImageSourceConverter x:Key="ResourceImageSourceConverter" />
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

Damit sind die vorbereitenden Arbeiten für die Bilder abgeschlossen, sodass wir uns nun den Schriftarten zuwenden können.

Im Ordner *Fonts* gehen wir mit den folgenden Dateien analog zu den Bilddateien vor:

- *Font Awesome 5 Free-Solid-900.otf*
- *ShadowsIntoLight-Regular.ttf*
- *Sofia-Regular.ttf*

Auch hier setzen wir in den Dateieigenschaften die Eigenschaft *Buildvorgang* auf den Wert *Eingebettete Ressource*.

Innerhalb des Ordners befindet sich außerdem noch die Datei *FaSolid.cs*, deren Inhalt Sie in Listing 11.8 sehen. Ihr Inhalt wurde über die Webseite *https://andreinitescu.github.io/IconFont2Code* für die Schriftart *Font Awesome 5 Free-Solid-900.otf* generiert. Sie beinhaltet Konstanten für sämtliche Icons, die wir innerhalb der App nutzen werden.

Listing 11.8 Über die Konstanten der Klasse FaSolid können wir komfortabel Icons in unserer App nutzen

```
static class FaSolid
{
  public const string Home = "\uf015";
  public const string Utensils = "\uf2e7";
```

```
    public const string Carrot = "\uf787";
    public const string UtensilSpoon = "\uf2e5";
    public const string PepperHot = "\uf816";
    public const string IceCream = "\uf810";
    public const string GlassCheers = "\uf79f";
    public const string UserCircle = "\uf2bd";
    public const string MapMarkedAlt = "\uf5a0";
    public const string Phone = "\uf095";
}
```

Um auch die Schriftarten in der App nutzen zu können, registrieren wir sie in der Datei *AssemblyInfo.cs* wie folgt und schließen die vorbereitenden Arbeiten damit ab:

```
[assembly: ExportFont("Font Awesome 5 Free-Solid-900.otf", Alias = "Fa-Solid")]
[assembly: ExportFont("ShadowsIntoLight-Regular.ttf", Alias = "Shadows")]
[assembly: ExportFont("Sofia-Regular.ttf", Alias = "Sofia")]
```

11.4.1 Startseite und Menü

Nachdem die vorbereitenden Arbeiten nun abgeschlossen sind, nehmen wir uns als Erstes die Startseite und das Navigationsmenü vor. Unser Ziel ist es, das Layout aus Bild 11.10 zu kreieren.

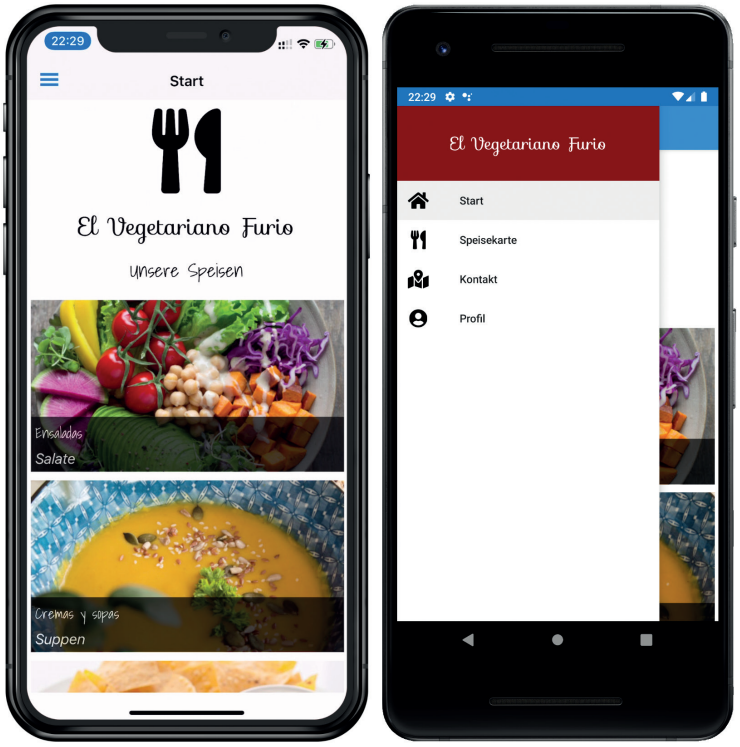

Bild 11.10 Die mit Schriftarten, Bildern und Icons angepasste Startseite und das Menü unserer App

Wie Sie der Abbildung entnehmen können, wirkt die Oberfläche durch die Verwendung von Bildern, eigenen Schriftarten und Icons direkt viel ansprechender.

Um das gewünschte Ziel zu erreichen, starten wir mit dem Menü in der Datei *AppShell.xaml* und ergänzen die fett gedruckten Zeilen aus Listing 11.9. Die Änderungen beschränken sich auf die angepasste Schriftart *Sofia* für die Überschrift *El Vegetariano Furio* und die Angabe von Icons für die Menüeinträge über mehrere `FontImageSource`-Anweisungen, die alle fett gedruckt sind.

Listing 11.9 Die angepasste Shell AppShell.xaml

```
<Shell … Namespacedeklarationen zur besseren Übersichtlichkeit entfernt …>
  <Shell.FlyoutHeaderTemplate>
    <DataTemplate>
      <StackLayout Padding="30" BackgroundColor="DarkRed">
        <Label FontSize="Large"
             TextColor="White"
             FontAttributes="Bold"
             HorizontalOptions="Center"
             VerticalOptions="CenterAndExpand"
             FontFamily="Sofia">
          El Vegetariano Furio
        </Label>
      </StackLayout>
    </DataTemplate>
  </Shell.FlyoutHeaderTemplate>

  <ShellContent Title="Start" ContentTemplate="{DataTemplate local:MainPage}">
    <ShellContent.Icon>
      <FontImageSource FontFamily="Fa-Solid"
        Glyph="{x:Static fonts:FaSolid.Home}" Color="Black"></FontImageSource>
    </ShellContent.Icon>
  </ShellContent>

  <FlyoutItem Title="Speisekarte">
    <FlyoutItem.Icon>
      <FontImageSource FontFamily="Fa-Solid"
        Glyph="{x:Static fonts:FaSolid.Utensils}" Color="Black"></FontImageSource>
    </FlyoutItem.Icon>
    <ShellContent Title="Ensaladas">
      <ShellContent.Icon>
        <FontImageSource FontFamily="Fa-Solid"
          Glyph="{x:Static fonts:FaSolid.Carrot}"></FontImageSource>
      </ShellContent.Icon>
      <ShellContent.ContentTemplate> <!-- Inhalt zur besseren Übersichtlichkeit
         entfernt --> </ShellContent.ContentTemplate>
    </ShellContent>
    <ShellContent Title="Sopas">
      <ShellContent.Icon>
        <FontImageSource FontFamily="Fa-Solid"
           Glyph="{x:Static fonts:FaSolid.UtensilSpoon}"></FontImageSource>
      </ShellContent.Icon>
      <ShellContent.ContentTemplate> <!-- Inhalt zur besseren Übersichtlichkeit
        entfernt --> </ShellContent.ContentTemplate>
    </ShellContent>
    <ShellContent Title="Tapas">
      <ShellContent.Icon>
        <FontImageSource FontFamily="Fa-Solid"
```

```
            Glyph="{x:Static fonts:FaSolid.PepperHot}"></FontImageSource>
        </ShellContent.Icon>
        <ShellContent.ContentTemplate> <!-- Inhalt zur besseren Übersichtlichkeit
          entfernt --> </ShellContent.ContentTemplate>
      <ShellContent Title="Principales">
        <ShellContent.Icon>
          <FontImageSource FontFamily="Fa-Solid"
             Glyph="{x:Static fonts:FaSolid.Utensils}"></FontImageSource>
        </ShellContent.Icon>
        <ShellContent.ContentTemplate> <!-- Inhalt zur besseren Übersichtlichkeit
           entfernt --> </ShellContent.ContentTemplate>
      </ShellContent>
      <ShellContent Title="Postres">
        <ShellContent.Icon>
          <FontImageSource FontFamily="Fa-Solid"
            Glyph="{x:Static fonts:FaSolid.IceCream}"></FontImageSource>
        </ShellContent.Icon>
        <ShellContent.ContentTemplate> <!-- Inhalt zur besseren Übersichtlichkeit
          entfernt --> </ShellContent.ContentTemplate>
      </ShellContent>
      <ShellContent Title="Bebidas">
        <ShellContent.Icon>
          <FontImageSource FontFamily="Fa-Solid"
             Glyph="{x:Static fonts:FaSolid.GlassCheers}"></FontImageSource>
        </ShellContent.Icon>
        <ShellContent.ContentTemplate> <!-- Inhalt zur besseren Übersichtlichkeit
          entfernt --> </ShellContent.ContentTemplate>
      </ShellContent>
    </FlyoutItem>

    <ShellContent Title="Kontakt" ContentTemplate="{DataTemplate local:MainPage}">
      <ShellContent.Icon>
        <FontImageSource FontFamily="Fa-Solid"
          Glyph="{x:Static fonts:FaSolid.MapMarkedAlt}" Color="Black">
                </FontImageSource>
      </ShellContent.Icon>
    </ShellContent>
    <ShellContent Title="Profil" ContentTemplate="{DataTemplate profile:ProfilePage}">
      <ShellContent.Icon>
        <FontImageSource FontFamily="Fa-Solid"
          Glyph="{x:Static fonts:FaSolid.UserCircle}" Color="Black">
                </FontImageSource>
      </ShellContent.Icon>
    </ShellContent>
</Shell>
```

Es gab zwar viele Änderungen, jede für sich ist jedoch recht simpel. Ähnlich einfach verhält es sich mit der Startseite, deren Änderungen Sie fett gedruckt in Listing 11.10 sehen.

Listing 11.10 Die angepasste Startseite MainPage.xaml

```
<StackLayout>
  <Label Text="{x:Static fonts:FaSolid.Utensils}" FontFamily="Fa-Solid"
      FontSize="99" HorizontalTextAlignment="Center" Padding="10"></Label>
  <Label Text="El Vegetariano Furio" FontSize="Title"
    HorizontalTextAlignment="Center" FontFamily="Sofia"></Label>
  <Label FontSize="Large" HorizontalTextAlignment="Center" Padding="5"
   FontFamily="Shadows">Unsere Speisen</Label>
```

```
  <CollectionView ItemsSource="{Binding Categories}">
    <CollectionView.ItemTemplate>
      <DataTemplate x:DataType="menu:Category">
        <Grid Padding="5" HeightRequest="200">
          <Image HeightRequest="200" Aspect="AspectFill"
           HorizontalOptions="FillAndExpand"
           Source="{Binding Id,
             Converter={StaticResource ResourceImageSourceConverter},
             ConverterParameter='ElVegetarianoFurio.Images.Categories.{0}.jpg'}"/>
          <StackLayout Padding="5" VerticalOptions="End"
             BackgroundColor="Black" Opacity="0.7">
            <Label Text="{Binding Name}" FontFamily="Shadows" TextColor="White">
            </Label>
            <Label Text="{Binding Description}" FontAttributes="Italic"
              TextColor="White"></Label>
          </StackLayout>
          <!-- Inhalt zur besseren Übersichtlichkeit entfernt -->

        </Grid>
      </DataTemplate>
    </CollectionView.ItemTemplate>
  </CollectionView>
</StackLayout>
```

Innerhalb des Listings starten wir mit dem Hinzufügen des Besteck-Icons. Dies geschieht über das erste Label innerhalb des StackLayouts. Wir geben dort an, dass wir die Schriftart *Fa-Solid* nutzen möchten, also eine Symbolschriftart. Aus dieser Schriftart zeigen wir in Schriftgröße 99 das Symbol *Utensils* an, welches für Besteck steht. Anschließend definieren wir für die beiden folgenden Labels die Nutzung der Schriftarten *Sofia* bzw. *Shadows*.

Etwas interessanter wird es innerhalb des DataTemplates der CollectionView. Hier ersetzen wir das bisherige Platzhalterrechteck, das über eine BoxView gerendert wurde, durch ein Bild. Mithilfe der zuvor beschriebenen Klasse `ResourceImageSourceConverter` binden wir die ID der jeweiligen Kategorie an die Eigenschaft `Source` des `Image`-Elements und erzeugen somit in Kombination mit dem `ConverterParameter` einen gültigen Pfad zu einem eingebetteten Bild.

Zu guter Letzt ändern wir noch die Schriftart der Kategorieüberschrift innerhalb des CollectionView-DataTemplates auf die Schriftart `Shadows`. Damit sind die Änderungen an der Startseite und am Menü abgeschlossen.

11.4.2 Kategorie- und Detailseite

Die Änderungen sowohl an der Kategorie- als auch an der Detailseite beschränken sich auf den Einsatz von Bildern und einer angepassten Schriftart. Wie Sie in Bild 11.11 sehen, zeigen diese Änderungen jedoch große Wirkung.

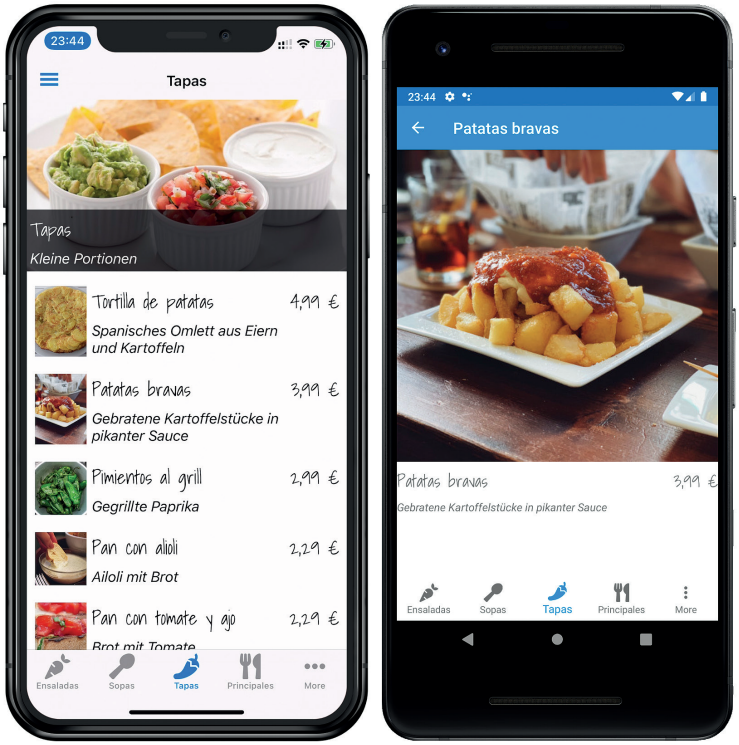

Bild 11.11 Die neue Kategorieseite unter iOS und die neue Detailseite unter Android

Die Anpassungen an beiden Seiten sind sehr ähnlich zu den Anpassungen auf der Startseite. Die Platzhalter werden durch Bilder ersetzt und über den Einsatz von Schriftarten wird die Oberfläche attraktiver gestaltet. Aufgrund der vielen Überschneidungen zur Startseite aus Listing 11.10 folgen an dieser Stelle Listing 11.11 für die Kategorieseite und Listing 11.12 für die Detailseite ohne weitere Erklärung. Die Änderungen sind wieder fett gedruckt.

Listing 11.11 Die geänderte Kategorieseite CategoryPage.xaml

```
<Grid RowDefinitions="Auto,*">
  <Image HeightRequest="200" Aspect="AspectFill" HorizontalOptions="FillAndExpand"
    Source="{Binding Category.Id,
    Converter={StaticResource ResourceImageSourceConverter},
    ConverterParameter='ElVegetarianoFurio.Images.Categories.{0}.jpg'}"/>
  <StackLayout Padding="5" VerticalOptions="End" BackgroundColor="Black"
     Opacity="0.7">
    <Label Text="{Binding Category.Name}" FontSize="Large" FontAttributes="Bold"
     TextColor="White" FontFamily="Shadows"></Label>
    <Label Text="{Binding Category.Description}"
     FontAttributes="Italic" TextColor="White"></Label>
  </StackLayout>

  <CollectionView Grid.Row="1" ItemsSource="{Binding Dishes}">
    <CollectionView.ItemTemplate>
```

```
      <DataTemplate x:DataType="menu:Dish">
        <Grid Padding="10" RowDefinitions="Auto, Auto"
           ColumnDefinitions="60, *, Auto">
          <Image HeightRequest="60" Grid.RowSpan="2" Aspect="AspectFill"
             HorizontalOptions="FillAndExpand"
             Source="{Binding Id,
             Converter={StaticResource ResourceImageSourceConverter},
             ConverterParameter='ElVegetarianoFurio.Images.Dishes.{0}.jpg'}"/>
          <Label Grid.Column="1" Text="{Binding Name}"
            FontFamily="Shadows" FontSize="Large"></Label>
          <Label Grid.Column="2" Text="{Binding Price, StringFormat='{0:C}'}"
            FontFamily="Shadows" FontSize="Large"></Label>
          <Label Grid.Row="1" Grid.Column="1" Text="{Binding Description}"
             FontAttributes="Italic"></Label>
          <!-- Inhalt zur besseren Übersichtlichkeit entfernt -->
        </Grid>
      </DataTemplate>
    </CollectionView.ItemTemplate>
  </CollectionView>
</Grid>
```

Listing 11.12 Die angepasste Detailseite DishPage.xaml

```
<StackLayout>
  <Image HeightRequest="400" Aspect="AspectFill" HorizontalOptions="FillAndExpand"
    Source="{Binding Dish.Id,
    Converter={StaticResource ResourceImageSourceConverter},
    ConverterParameter='ElVegetarianoFurio.Images.Dishes.{0}.jpg'}"/>
  <Grid RowDefinitions="Auto,Auto" ColumnDefinitions="*,Auto">
    <Label Text="{Binding Dish.Name}" FontSize="Large" FontFamily="Shadows"></Label>
    <Label Grid.Column="1" Text="{Binding Dish.Price, StringFormat='{0:C}'}"
     FontSize="Large" FontFamily="Shadows"></Label>
    <Label Grid.Row="1" Text="{Binding Dish.Description}"
      FontAttributes="Italic"></Label>
  </Grid>
</StackLayout>
```

Mit diesen Änderungen ist unsere Oberflächenoptimierung abgeschlossen. Wie Sie gesehen haben, lässt sich bereits mit geringem Aufwand eine große Wirkung erzielen.

■ 11.5 Was Sie in diesem Kapitel gelernt haben

In diesem Kapitel haben wir uns mit Bildern und Schriftarten beschäftigt. Sie haben gelernt, dass Bilder plattformspezifisch oder plattformübergreifend dargestellt werden. Die plattformspezifische Darstellung ist die aufwendigere Variante, aber auch die, mit der Sie das beste Ergebnis erzielen können. Dadurch, dass die Bilder plattformspezifisch in mehreren Auflösungen bereitgestellt werden, können Sie sogar unterschiedliche Abbildungen je Plattform verwenden.

Bei der plattformübergreifenden Bildanzeige haben Sie etwas weniger Flexibilität. Unter anderem nutzen Sie immer dasselbe Bild mit einer einzigen Auflösung. Welche Variante für Ihre App die beste ist, lässt sich nicht pauschal beantworten. Ich selbst versuche stets, die plattformübergreifende Variante zu nehmen und greife nur bei hohen Anforderungen zur plattformspezifischen Variante.

Im zweiten Teil des Kapitels haben wir uns mit Schriftarten beschäftigt. Sie haben in diesem Teil des Kapitels gesehen, dass benutzerdefinierte Schriftarten mit wenig Aufwand genutzt werden können. Außerdem haben Sie gelernt, dass Sie mithilfe von Symbolschriftarten Icons in beliebiger Größe und Farbe in Ihrer App darstellen können.

Register

E

F

G

H

I

J

K

T

U

V

W

X