

Teil G

# Windows Store-Apps

**In diesem Teil:**

Einführung in Windows Store-Apps	803
Layouts und Steuerelemente	829
Spezielle Techniken	867
Von der Idee zum Windows Store	903



## Kapitel 33

# Einführung in Windows Store-Apps

### **In diesem Kapitel:**

Einführung	804
Die erste Windows Store-App	807

Mit der Einführung von Windows 8 hat Microsoft begonnen, den klassischen PC-Desktop mit der schillernden Welt der Tablets und Smartphones zu verschmelzen. Seitdem heißen alle Anwendungen, die auf Windows laufen, offiziell *Apps* und werden in zwei Gruppen klassifiziert: die Desktop-Apps, hinter denen sich letztlich die klassischen Desktop-Anwendungen verbergen, und die Windows Store-Apps, welche für die neue Touchscreen-freundliche Windows-Benutzeroberfläche entwickelt werden. Was sich hinter diesen Windows Store-Apps verbirgt, was sie auszeichnet und wie Sie mit Visual Studio Windows Store-Apps erstellen, ist Thema dieses Kapitels.

## Einführung

Eine Anwendung, die für die neue Windows-Benutzeroberfläche geschrieben wird, heißt offiziell *Windows Store-App*. Windows Store-Apps können über den Windows Store vertrieben werden, sie müssen es aber nicht. Anders ausgedrückt: Windows Store-Apps haben nicht zwangsweise etwas mit dem Windows Store zu tun, sondern sind einfach nur Apps, die dem neuen Windows-App-Design folgen und dem neuen WinRT-Ausführungsmodell unterliegen – weswegen sie auch als *WinRT-Apps* bezeichnet werden.

Wie dieses Design und das Ausführungsmodell aussehen, soll im Folgenden kurz skizziert werden.

## Neues Design

Was bereits bei einem flüchtigen Blick auf eine Windows Store-App direkt auffällt, sind das flache, verzierungslose Erscheinungsbild und die – für Tablet- und Smartphone-Anwendungen typische – Ausdehnung über den gesamten Bildschirm.

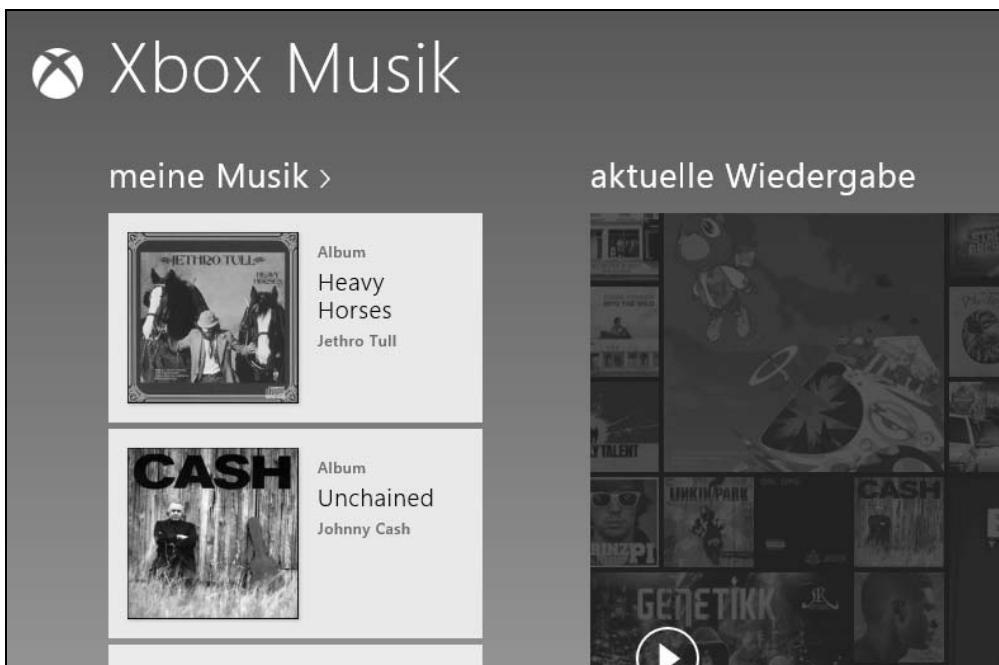


Abbildung 33.1 Die Musik-App von Windows 8

Windows Store-Apps weisen die folgenden Design-Merkmale auf:

- **bildschirmfüllend** Windows Store-Apps nehmen grundsätzlich den gesamten Bildschirm ein. Da dies sowohl der Monitor eines PCs als auch der Touchscreen eines Tablets oder Smartphones sein kann, sollte eine App unter unterschiedlichsten Bildschirmauflösungen (inklusive Snap-Modus) sinnvoll angezeigt werden und gut bedienbar sein.

Zur besseren Unterstützung verschiedener Bildschirmauflösungen sollten Sie beim Aufbau der Benutzeroberfläche absolute Dimensionierung und Positionierung wo sinnvoll vermeiden und stattdessen die Möglichkeiten der verschiedenen Layout-Container (Grid, StackPanel etc.) und Layout-Eigenschaften (HorizontalAlignment, Padding etc.) nutzen (siehe Kapitel 34 und 29).

Zur direkten Kontrolle Ihres Designs können Sie das *Gerät*-Fenster verwenden (siehe Kapitel 34).

- **seitenbasiert** Statt aus Fenstern und Dialogfeldern besteht die Benutzeroberfläche einer Windows Store-App aus einer oder mehreren bildschirmfüllenden Seiten, die in flacher Hierarchie organisiert sein sollten und von denen üblicherweise jede eine klar definierte Aufgabe übernimmt.

Visual Studio bietet diverse Vorlagen an, die Sie für den Aufbau Ihrer Seiten nutzen können (siehe Abschnitt »Die erste Windows Store-App«).

Wie Sie Navigationselemente zum Wechseln zwischen den Seiten einer App anlegen, lesen Sie in Kapitel 35. Dort erfahren Sie auch, wie Sie vordefinierte Seitenbereiche dynamisch in eine übergeordnete Seite einblenden.

- **fingergesteuert** Die Seiten einer Windows Store-App sollten aus eher großflächigen Elementen aufgebaut werden und gut durch Fingergesten steuerbar sein. Wie sie auf Fingergesten reagieren, lesen Sie später in diesem Kapitel in dem Abschnitt, der sich mit der Ereignisbehandlung beschäftigt.
- **schmucklos** Microsoft propagiert derzeit App-Designs, die möglichst ganz ohne Dekorationen (Menüleisten, Symbolleisten), unnötigen »Schnickschnack« oder simulierter 3D-Optik auskommen. Was allerdings erwünscht ist, ist der Einsatz von Animationen für Seitenübergänge und andere Aktionen, um dem Benutzer ein möglichst fließendes und natürliches Erlebnis bei der Verwendung der App zu vermitteln (siehe Kapitel 34).

- **menülos** Windows Store-Apps verfügen (mit Ausnahme eines optionalen Header-Menüs) über keine Menüleiste (Kontextmenüs können notfalls mithilfe von `PopupMenu` realisiert werden).

Ideal wäre es, wenn alle wichtigen Aktionen direkt über Interaktion mit den Seitenelementen ausgelöst werden können (siehe später in diesem Kapitel den Abschnitt, der sich mit der Ereignisbehandlung beschäftigt).

Bestimmte Befehle, wie z. B. das Drucken, können über die vordefinierten Charms (siehe Kapitel 35) zur Verfügung gestellt werden. Und für die restlichen Befehle gibt es als Menüleistenersatz die Befehlsleiste (siehe Kapitel 34).

## Neue Laufzeitumgebung

Windows Store-Apps haben ihr eigenes Ausführungsparadigma, das auf die Windows Runtime, kurz WinRT, zurückgeht (siehe auch Kapitel 1). Dieses Ausführungsmodell unterscheidet sich in einigen Punkten ganz erheblich von dem für Desktop-Anwendungen üblichen Paradigma:

- **Sandbox-Sicherheitsverwahrung** WinRT-Anwendungen werden, ausgestattet mit bestimmten Grundrechten (Base Trust), in ihrer eigenen lokalen Umgebung, der so genannten *Sandbox*, ausgeführt. Dies erhöht die Sicherheit und sorgt dafür, dass abstürzende Apps keine anderen Apps mitreißen können. Zugriffe auf geschützte Bereiche und Ressourcen sind nur durch die Vermittlung spezieller Broker, wie z. B. den `FilePicker`-Objekten (siehe Kapitel 35), möglich. Möchte die App mit Fähigkeiten und Rechten ausgestattet werden, die über die Base Trust-Rechte hinausgehen, muss sie dies anmelden (siehe Kapitel 35) – und gegebenenfalls darauf hoffen, dass der Anwender ihr die gewünschten Rechte bei der Ausführung zugesteht.
- **Lebenszyklus** Windows Store-Apps haben ihren eigenen Lebenszyklus, der sich gegenüber dem Lebenszyklus von Desktop-Anwendungen vor allem dadurch auszeichnet, dass in den Hintergrund tretende Apps (Suspended-Modus genannt) komplett vom Prozessor getrennt werden. Mit anderen Worten: Nur die im Vordergrund befindliche App bekommt vom Betriebssystem Prozessorzeit zugeteilt und nur diese App verbraucht Strom und Prozessorleistung. Windows Store-Apps sollten dies grundsätzlich respektieren und z. B. ausgeführte Hintergrundaktivitäten beim Eintritt in den Suspended-Modus anhalten. (Mehr zum Lebenszyklus in Kapitel 35.)

Da Windows Store-Apps im Hintergrund (Suspended-Modus) keine Prozessorzeit beanspruchen, ja bei Ressourcenengpässen sogar automatisch vom Windows-Betriebssystem ganz beendet werden können, ist es in der Regel nicht mehr nötig, dass der Anwender einmal gestartete Apps explizit beendet. Windows Store-Apps besitzen daher üblicherweise keinen Beenden-Befehl. (Sie können aber durch die entsprechende Fingergeste oder durch Drücken der Tastenkombination `[Alt] + [F4]`-Kombination beendet werden. Um eine App vom Code aus zu beenden, gibt es die `Exit()`-Methode des `Application`-Objekts).

- **Sterben statt einfrieren** Nachlässig programmierte Desktop-Anwendungen leiden nicht selten unter dem Einfrieren ihrer Benutzeroberfläche – ausgelöst durch rechenintensive oder länger andauernde Aktionen, die im gleichen Thread wie die Benutzeroberfläche (UI-Thread) ausgeführt werden und dadurch sowohl die Aktualisierung der Oberfläche als auch die Verarbeitung weiterer Benutzerereignisse verhindern. In Windows Store-Apps müssen Sie noch stärker als bei Desktop-Anwendungen darauf achten, dass die App nie ihre Reaktionsfähigkeit verliert – nicht nur weil entsprechende Fehlerquellen (wie z. B. das Warten auf Daten aus dem Internet) in Apps viel häufiger anzutreffen sind, sondern auch weil das Betriebssystem nicht mehr reagierende Apps beenden kann. Ihre wichtigsten Hilfsmittel beim Kampf gegen das Einfrieren sind die asynchrone Ausführung (siehe im Unterkapitel »Die erste Windows Store-App« den Abschnitt zur asynchronen Programmierung und in Kapitel 23 den Abschnitt zu `async` und `await`), die Auslagerung rechenintensiver Operationen in Threads oder Tasks (siehe Kapitel 38 und 39) sowie – als letzte Absicherung – das Speichern des Anwendungszustands beim Übergang in den Suspended-Modus (siehe Kapitel 35).

---

**HINWEIS** Für die Entwicklung von Windows Store-Apps stehen nicht mehr alle Klassen aus dem .NET Framework zur Verfügung. Insbesondere die Klassen und Namespaces für den Aufbau von Benutzeroberflächen (wie ASP.NET oder WPF) wurden durch die neuen Klassen aus dem Namespace `Windows` ersetzt. Geblieben sind vor allem Utility-Klassen für grundlegende Funktionen, beispielsweise aus den Namespaces `System`, `System.Collections.Generics`, `System.ComponentModel`, `System.Diagnostics`, `System.Globalization`, `System.IO`, `System.Linq`, `System.Net`, `System.Reflection`, `System.Text`, `System.Threading`, `System.Threading.Tasks`, `System.XML`.

---

## Einpassung in die neue Windows-Benutzeroberfläche

Als Teil der neuen Windows-Benutzeroberfläche kann jede Windows Store-App von folgenden Features profitieren:

- Kacheln (siehe Kapitel 34)
- Begrüßungsbildschirm (siehe Kapitel 34)
- Befehlsleiste (siehe Kapitel 34)
- Charms (siehe Kapitel 35)
- Vertrieb über den Windows Store (siehe Kapitel 36)

## Die erste Windows Store-App

Zur Eingewöhnung in die Programmierung von Windows Store-Apps werden wir in den folgenden Abschnitten ein Projekt für eine erste App anlegen, analysieren und schrittweise ausbauen.

### Entwicklerlizenz

Um Windows Store-Apps entwickeln und testen zu können, benötigen Sie eine Entwicklerlizenz. Diese ist kostenlos, gilt aber nur für die Entwicklung und das Testen von Apps auf dem registrierten Rechner. Die Entwicklerlizenz wird nur für einen kurzen fixen Zeitraum vergeben, kann aber bei Bedarf jederzeit erneuert werden.

Wenn Sie mit Visual Studio das erste Mal ein Projekt für eine Windows Store-App anlegen, springt automatisch ein Dialogfeld auf, über das Sie eine Entwicklerlizenz erwerben können. Sie benötigen für den Erwerb der Lizenz allerdings ein Microsoft-Konto. Dieses können Sie bei Bedarf zwar ebenfalls kostenlos anlegen, sollten aber bedenken, dass über ein Microsoft-Konto auch kostenpflichtige Dienste in Anspruch genommen werden können. Vergeben Sie also unbedingt ein sicheres Passwort.

Das Dialogfeld zum Erwerb oder zur Verlängerung der Entwicklerlizenz können Sie natürlich auch jederzeit über Visual Studio aufrufen. Den entsprechenden Befehl finden Sie im Menü *Projekt/Store*.

---

**HINWEIS** Der Sicherheitsmechanismus von Windows 8 sorgt dafür, dass auf einem Rechner nur zertifizierte Apps ausgeführt werden. Apps, die über den Windows Store vertrieben werden, werden automatisch zertifiziert (siehe auch Kapitel 36). Die Zertifizierung über den Windows Store wäre für das Testen von in der Entwicklungsphase befindlichen Apps allerdings recht umständlich. Die Entwicklerlizenz ermöglicht Ihnen daher, Apps auf Ihrem lokalen Rechner ohne Zertifizierung auszuführen. (Eine dritte Form der Zertifizierung ist das Querladen von Unternehmens-Apps.)

---

## Schritt 1: Projekt anlegen

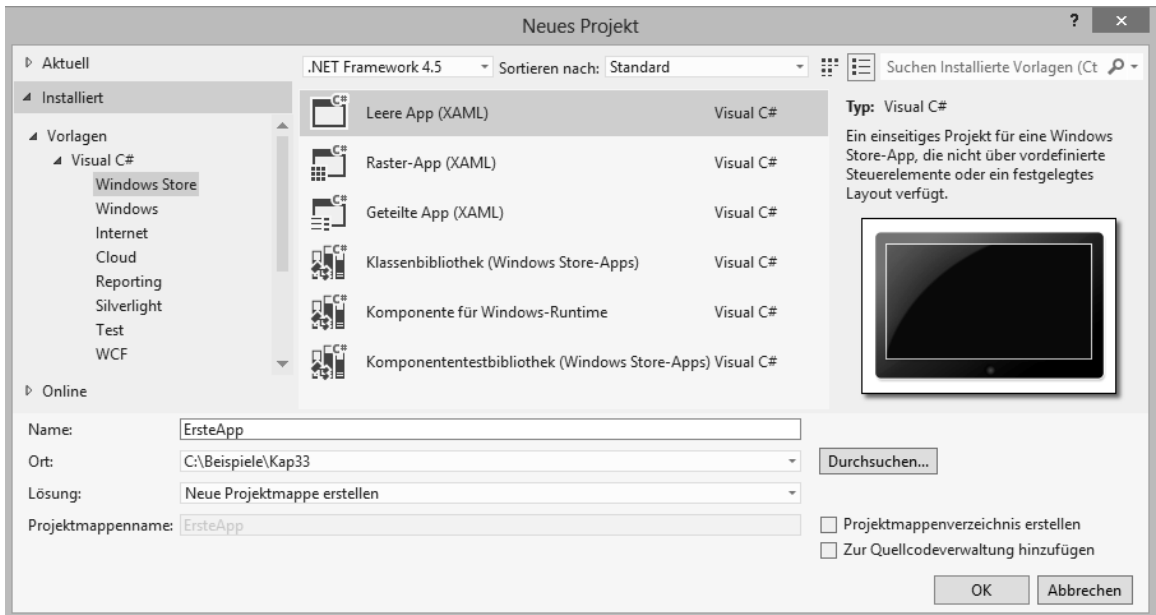


Abbildung 33.2 Anlegen eines neuen Windows Store-Projekts

1. Rufen Sie im Menü *Datei* den Befehl *Neu/Projekt* auf.
2. Wählen Sie im daraufhin angezeigten Dialogfeld *Neues Projekt* die Vorlagenkategorie *Visual C#/Windows Store* und anschließend als Vorlage *Leere App (XAML)* aus.
3. Geben Sie wie üblich einen Namen für das Projekt an, legen Sie den Speicherort fest und klicken Sie dann auf *OK*.

Projektvorlage	Beschreibung
Leere App	<p>Projekt mit einer einzigen, leeren Seite; ist aber auch für mehrseitige Projekte gut geeignet.</p> <p>Im Gegensatz zu den beiden anderen Projektvorlagen, bietet diese Vorlage nur ein ganz rudimentäres App-Grundgerüst. Insbesondere fehlt der App jedwede Unterstützung für die Einrichtung einer Navigationsstruktur – die für Apps mit nur einer Seite ja auch nicht benötigt wird.</p> <p>Trotzdem eignet sich diese Projektvorlage auch hervorragend für die Entwicklung mehrseitiger Apps, für welche die beiden anderen Vorlagen nicht in Frage kommen, weil sie zu stark spezialisiert sind. Sie müssen lediglich die mit der Vorlage installierte leere Seite durch eine andere Seite ersetzen, die auf einer Seitenvorlage mit Navigationsstruktur basiert (siehe weiter unten den Abschnitt »Startseite austauschen«).</p>



Projektvorlage	Beschreibung
Geteilte App	<p>Projektvorlage für eine zweistufige Hierarchie zur Repräsentation von Datenelementen, die sich in Gruppen klassifizieren lassen. Die Startseite <i>ItemsPage</i> ist dafür gedacht, die Gruppen anzuzeigen und zur Auswahl anzubieten. Tippt der Anwender auf eine Gruppe, wird er zur zweiten Seite geführt (<i>SplitPage</i>), die zweigeteilt ist: Links werden die in der Gruppe enthaltenen Elemente aufgelistet, rechts werden Details zu dem links ausgewählten Element angezeigt.</p> <p>Neben einer vorgefertigten Navigationsstruktur und einem Layout, das bereits für die Darstellung unter verschiedenen Auflösungen und Formaten ausgelegt ist, bringt die Vorlage auch bereits ein passendes Datenmodell mit (siehe Datei <i>SampleDataSource.cs</i> im Projektordner <i>DataModel</i>).</p> <p>Layout und Datenmodell gehen von Datenelementen aus, die über ein Bild (<i>Image</i>), einen Titel (<i>Title</i>), einen Untertitel (<i>Subtitle</i>), eine Beschreibung (<i>Description</i>) und Inhalt (<i>Content</i>) verfügen.</p> <p>Wenn Sie Glück haben und sich Ihre Daten an das vorgegebene Datenmodell anpassen lassen (d.h. die Klassen, die Ihre Daten repräsentieren, lassen sich nach dem Vorbild der Datenklassen in der Datei <i>SampleDataSource.cs</i> aufbauen), können Sie Layout und Datenmodell direkt übernehmen. Ansonsten steht es Ihnen frei, Layout und Datenmodell anzupassen oder ganz zu ersetzen. Im letzteren Fall stellt sich dann allerdings die Frage, ob es nicht weniger aufwändig wäre, das Projekt auf der Vorlage <i>Leere App</i> aufzubauen.</p>
Raster-App	<p>Projektvorlage für eine dreistufige Hierarchie zur Repräsentation von Datenelementen, die sich in Gruppen klassifizieren lassen. Die <i>Raster-App</i>-Vorlage ist im Grunde eine Erweiterung der Vorlage <i>Geteilte App</i>, bei der aus der zweigeteilten Ansicht der Detailseite zwei getrennte Seiten wurden.</p> <p>Die Startseite <i>GroupedItemsPage</i> ist dafür gedacht, die Gruppen mit ihren Elementen anzuzeigen und zur Auswahl anzubieten. Tippt der Anwender hier auf den Titel einer Gruppe, wird er zur zweiten Seite geführt (<i>GroupDetailPage</i>), die eine Gruppenbeschreibung und eine Liste der in der Gruppe enthaltenen Elemente enthält. Tippt der Anwender hier auf eines der enthaltenen Elemente, wird er zur dritten Seite geführt (<i>ItemDetailPage</i>), die Details zu dem ausgewählten Element anzeigt. Alternativ kann der Anwender die dritte Seite aber direkt von der Startseite aus erreichen, indem er statt auf einen Gruppentitel auf eines der angezeigten Elemente tippt.</p> <p>Neben einer vorgefertigten Navigationsstruktur und einem Layout, das bereits für die Darstellung unter verschiedenen Auflösungen und Formaten ausgelegt ist, bringt die Vorlage auch bereits ein passendes Datenmodell mit (siehe Datei <i>SampleDataSource.cs</i> im Projektordner <i>DataModel</i>).</p> <p>Das verwendete Datenmodell ist identisch zu dem Datenmodell der Vorlage <i>Geteilte App</i> und geht von Datenelementen aus, die über ein Bild (<i>Image</i>), einen Titel (<i>Title</i>), einen Untertitel (<i>Subtitle</i>), eine Beschreibung (<i>Description</i>) und Inhalt (<i>Content</i>) verfügen.</p> <p>Wenn Sie Glück haben und sich Ihre Daten an das vorgegebene Datenmodell anpassen lassen (d.h. die Klassen, die Ihre Daten repräsentieren, lassen sich nach dem Vorbild der Datenklassen in der Datei <i>SampleDataSource.cs</i> aufbauen), können Sie Layout und Datenmodell direkt übernehmen. Ansonsten steht es Ihnen frei, Layout und Datenmodell anzupassen oder ganz zu ersetzen. Im letzteren Fall stellt sich dann allerdings die Frage, ob es nicht weniger aufwändig wäre, das Projekt auf der Vorlage <i>Leere App</i> aufzubauen.</p>

**Tabelle 33.1** Die mit Visual Studio installierten Projekte für Windows Store-Apps

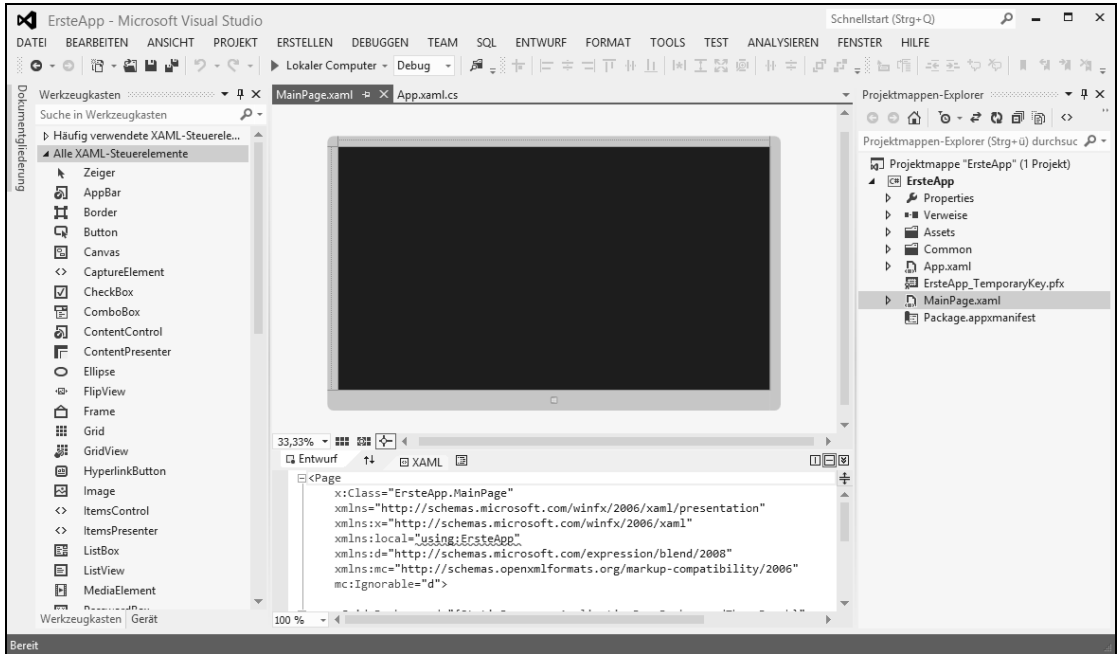


Abbildung 33.3 Das neue Projekt in der Visual Studio-Umgebung

## Die Projektdateien

Selbst ein rudimentäres Projekt, wie es die Projektvorlage *Leere App* erzeugt, beinhaltet bereits eine ganze Reihe von Dateien, die nötig sind, um aus dem Projekt eine funktionierende Windows Store-App werden zu lassen. Tabelle 33.3 stellt Ihnen die wichtigsten dieser Dateien vor. Zuvor aber werden wir einen detaillierten Blick auf diejenigen Dateien werfen, die den Code unserer App und ihrer Startseite beherbergen:

- *App.xaml* und *App.xaml.cs* enthalten den Code der App-Anwendung
- *MainPage.xaml* und *MainPage.xaml.cs* enthalten den Code der Startseite

Wie im Falle von WPF wird also auch bei Windows Store-Apps die Benutzeroberfläche als Kombination aus XAML-Code und C#-Code definiert. Für den App-spezifischen Code und für jede Seite der App gibt es daher sowohl eine XAML-Datei, die den XAML-Code enthält, als auch eine Code-Behind-Datei mit dem zugehörigen C#-Code.

**HINWEIS** Die Aufteilung in XAML- und C#-Code dient dem Zweck, Layout (XAML-Code) und Logik (C#-Code) zu trennen (siehe auch den Abschnitt »Grundprinzipien« und dort den Unterabschnitt »Trennung von Design und Logik« in Kapitel 28).

## MainPage.xaml

Doppelklicken Sie nun im Projektmappen-Explorer auf den Knoten der *MainPage.xaml*-Datei, um sie in den Designer zu laden. Im oberen Entwurf-Bereich sehen Sie eine grafische Darstellung des leeren Fensters, darunter im XAML-Bereich steht der zugehörige XAML-Code, der Aussehen und Layout des Fensters definiert:

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

  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">

  </Grid>
</Page>
```

Listing 33.1 *MainPage.xaml*

## XAML-Superschnellkurs

Vielen Lesern wird XAML bereits von anderen Technologien (wie z.B. WPF, siehe Teil F) bekannt sein. Für alle anderen hier eine (extrem) kurze Einführung:

Eine XAML-Datei definiert eine Elementhierarchie, an deren Kopf – wie es sich für wohlgeformtes XML gehört – ein einzelnes Wurzelement steht, hier `<Page>`. Woher stammt dieses Wurzelement? Der Elementname legt die Vermutung nahe, dass es sich um ein vordefiniertes Element handelt. Doch dies ist nur zum Teil richtig. XAML selbst definiert nämlich keine eigenen Elemente. Trotzdem ist der Name `Page` nicht frei erfunden, er leitet sich von der WinRT-Klasse `Page` ab. Um die Verbindung zwischen dem Bezeichner `Page` und der zugehörigen WinRT-Klasse herzustellen, muss der Namespace `http://schemas.microsoft.com/winfx/2006/xaml/presentation`<sup>1</sup> eingebunden werden.

Die meisten XAML-Dateien binden als zweiten Namespace `http://schemas.microsoft.com/winfx/2006/xaml` für die vordefinierten XAML-Attribute wie `Name`, `FieldModifier` etc. ein. Als Präfix wird für diesen Namespace per Übereinkunft `x:` verwendet. Wenn Sie Namen aus weiteren Namespaces verwenden möchten, müssen diese mit einer ID deklariert werden, die an das XML-Attribut `xmlns` anzuhängen ist. Wo immer später ein Name aus einem solchen Namespace verwendet wird, muss dem Namen die ID des Namespace als Präfix vorangestellt werden.

Eingebettet in das Wurzelement folgen die weiteren Elemente, die den Aufbau der Seite definieren. Hier sind alle WinRT-UI-Elemente erlaubt (siehe z.B. die Liste der XAML-Steuerelemente im Werkzeugkasten). All diese Elemente müssen in wohlgeformtem XML definiert werden, d.h. insbesondere die Werte, die den Attributen (entsprechen den `public`-Eigenschaften der zugehörigen WinRT-Klassen) zugewiesen werden, müssen in Anführungszeichen stehen, und die Elemente selbst müssen entweder aus Start- und End-Tag bestehen oder mit `/>` abgeschlossen werden:

---

<sup>1</sup> Der URL dient hier allein als eindeutiger Bezeichner für den Namespace.

```

<Page ... >
  <StackPanel Orientation="Vertical" Background="Black">
    <TextBlock Text="Bitte klicken:" FontSize="50" />
    <Button FontSize="50" Margin="50,20,20,0" Width="250">Aktion 1</Button>
    <Button FontSize="50" Margin="50,20,20,0" Width="250">Aktion 2</Button>
    <Button FontSize="50" Margin="50,20,20,0" Width="250">Aktion 3</Button>
  </StackPanel>
</Page>

```

Der obige Code definiert z.B. ein StackPanel-Element mit drei eingebetteten Schaltflächen (Button-Elemente). Das StackPanel-Element ist ein Layout-Container, dessen Aufgabe es ist, die in ihm eingebetteten Elemente auszurichten. Hier ordnet der StackPanel-Container die eingebetteten drei Schaltflächen senkrecht untereinander an (siehe den Wert des Attributs Orientation).

**HINWEIS** Verglichen mit WPF-Anwendungen ist es für Windows Store-Apps noch wichtiger, dass sich das Layout der Benutzeroberfläche an unterschiedliche Abmessungen (Monitorauflösung, Hoch- oder Querformat, Snapped-Modus) anpassen kann. In der Regel ist daher die Dimensionierung und Positionierung von Steuerelementen über Layout-Container und relative Größenangaben der Vergabe absoluter Werte vorzuziehen (siehe Schritt 3 und den Abschnitt »Seiten-Layout« in Kapitel 34).

## MainPage.xaml.cs

Um die Code-Behind-Datei zur Seite *MainPage* zu laden, können Sie wahlweise so vorgehen, dass Sie bei geladener *MainPage.xaml*-Datei im *Entwurf*-Bereich des Designers mit der rechten Maustaste in die dargestellte Seite klicken und im Kontextmenü den Befehl *Code anzeigen* aufrufen, oder im Projektmappen-Explorer den *MainPage.xaml*-Knoten expandieren und auf den Knoten der Code-Behind-Datei doppelklicken.

```

using System;
...
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

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

        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
        }
    }
}

```

Listing 33.2 *MainPage.xaml.cs*

Der C#-Code der *MainPage*-Seite der Projektvorlage *Leere App* ist relativ kurz und enthält anfangs noch keinerlei funktionalen Code, und doch gibt es einige Punkte, auf die ich kurz Ihre Aufmerksamkeit lenken möchte:

- die Einbindung der wichtigsten WinRT-Namespaces (beginnend mit *Windows*.)
- die Ableitung der Startseite von der WinRT-Klasse *Page*

Beachten Sie, dass die Basisklasse die gleiche Klasse ist, die in *MainPage.xaml* als Wurzelement der XAML-Hierarchie angegeben wurde. Wenn Sie die Basisklasse im C#-Code austauschen, müssen Sie auch das Wurzelement im XAML-Code umbenennen.

- der Aufruf der Methode `InitializeComponent()`

Die Methode `InitializeComponent()` ist in der Datei *MainPage.g.i.cs* definiert. Der Code dieser Datei sollte nur vom Visual Studio-Designer bearbeitet werden; Sie können ihn aber einsehen, indem Sie beispielsweise mit der rechten Maustaste in den Namen der `InitializeComponent()`-Methode klicken und den Befehl *Gehe zu Definition* auswählen.

Beachten Sie, dass für alle XAML-Elemente, denen Sie über das `Name`-Attribut einen Namen zuweisen (z.B.: `<Button Name="btn1">Aktion 1</Button>`), in der *MainPage.g.i.cs*-Datei ein Objekt instanziiert und unter besagtem Namen als Feld der Seite gespeichert wird:

```
partial class MainPage : global::Windows.UI.Xaml.Controls.Page
{
    ...
    private global::Windows.UI.Xaml.Controls.Button btn1;
    ...
    public void InitializeComponent()
    {
        ...

        btn1 = (global::Windows.UI.Xaml.Controls.Button)this.FindName("btn1");
    }
}
```

- die Überschreibung der *Page*-Methode `OnNavigatedTo()`

Diese Methode wird ausgeführt, wann immer die Seite angesteuert wird. Für Apps mit nur einer Seite wäre dies einmal beim Start der App. Für Apps mit mehreren Seiten wird die Methode dagegen jedes Mal ausgeführt, wenn die Seite auf den Bildschirm geholt wird.

Aktion	Aufrufabfolge
Seite wird aufgerufen	Konstruktor der Seite
	<code>OnNavigatedTo()</code> -Methode der Seite
	Loaded-Ereignis der Seite
Seite wird verlassen (Wechsel zu einer anderen Seite)	<code>OnNavigatingFrom()</code> -Methode der Seite
	<code>OnNavigatedFrom()</code> -Methode der Seite

**Tabelle 33.2** Aufrufe beim Ansteuern und Verlassen einer Seite

**HINWEIS** Mit die wichtigste Aufgabe der Code-Behind-Datei ist, die Definition der Ereignisbehandlungsmethoden aufzunehmen (siehe weiter unten den Schritt 3).

## App.xaml

Doppelklicken Sie nun im Projektmappen-Explorer auf den Knoten der *App.xaml*-Datei, um sie in den Editor zu laden.

Anders als die XAML-Dateien der App-Seiten definiert die Datei *App.xaml* keine UI-Elemente, die zum Aufbau der Benutzeroberfläche beitragen. Trotzdem hat sie mit dem Erscheinungsbild der Benutzeroberfläche zu tun. Man kann sie nämlich zur Definition von globalen Design-Stilen und -Ressourcen verwenden. Diese Stile können direkt in der *App.xaml*-Datei definiert werden oder aus einer externen XAML-Datei importiert werden. Letztere Technik nutzt die von der *Leere App*-Vorlage erzeugte *App.xaml*-Datei:

```
<Application
  x:Class="ErsteApp.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:ErsteApp">

  <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Common/StandardStyles.xaml"/>
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

Listing 33.3 *App.xaml*

## App.xaml.cs

Weit interessanter als der XAML-Part ist der Inhalt der Code-Behind-Datei der Anwendung.

```
using System;
...

namespace ErsteApp
{
  sealed partial class App : Application
  {
    public App()
    {
      this.InitializeComponent();
      this.Suspending += OnSuspending;
    }

    protected override void OnLaunched(LaunchActivatedEventArgs args)
    {
      Frame rootFrame = Window.Current.Content as Frame;
    }
  }
}
```

```

// App-Initialisierung nicht wiederholen, wenn das Fenster bereits Inhalte
// enthält. Nur sicherstellen, dass das Fenster aktiv ist.
if (rootFrame == null)
{
    // Einen Rahmen erstellen, der als Navigationskontext fungiert und zum
    // Parameter der ersten Seite navigieren
    rootFrame = new Frame();

    if (args.PreviousExecutionState == ApplicationExecutionState.Terminated)
    {
        //TODO: Zustand von zuvor angehaltener Anwendung laden
    }

    // Den Rahmen im aktuellen Fenster platzieren
    Window.Current.Content = rootFrame;
}

if (rootFrame.Content == null)
{
    // Wenn der Navigationsstapel nicht wiederhergestellt wird, zur ersten
    // Seite navigieren und die neue Seite konfigurieren, indem die
    // erforderlichen Informationen als Navigationsparameter übergeben werden
    if (!rootFrame.Navigate(typeof(MainPage), args.Arguments))
    {
        throw new Exception("Failed to create initial page");
    }
}
// Sicherstellen, dass das aktuelle Fenster aktiv ist
Window.Current.Activate();
}

private void OnSuspending(object sender, SuspendingEventArgs e)
{
    var deferral = e.SuspendingOperation.GetDeferral();
    //TODO: Anwendungszustand speichern und alle Hintergrundaktivitäten beenden
    deferral.Complete();
}
}
}

```

#### Listing 33.4 *App.xaml.cs*

In der Code-Behind-Datei der Anwendung wird der Lebenszyklus der App gesteuert. Unsere abgeleitete Klasse App erbt dazu von der Basisklasse Application diverse Ereignisse (Suspending und Resuming) und virtuelle Methoden (OnLaunched(), OnActivated() etc.). Wie Sie dem Code entnehmen können, wurde in der App-Klasse auch bereits ein rudimentäres Codegerüst zur Steuerung des App-Lebenszyklus angelegt:

- die virtuelle Methode OnLaunched() wurde überschrieben, um festzulegen, was beim Start der App geschehen soll
- das Suspending-Ereignis wurde mit einer Ereignisbehandlungsmethode verknüpft, in die Sie Code einfügen können, der ausgeführt werden soll, wenn die App in den Hintergrund tritt

Der App-Lebenszyklus wird glücklicherweise weitgehend automatisch gehandhabt, sodass wir eine detailliertere Einführung in den App-Lebenszyklus getrost auf Kapitel 35 verschieben können. Was uns im Moment weit mehr interessiert, ist der Code, der die Startseite der App zur Anzeige auf den Bildschirm bringt.

Dazu benötigt die App-Klasse ein Objekt der Klasse `Frame`. `Frame`-Objekte dienen als Container für Seiten (`Page`-Objekte) und bringen die Funktionalität zum Ansteuern und Wechseln zwischen Seiten mit. Unser einfaches *ErsteApp*-Projekt besteht zwar nur aus einer einzigen Seite, aber auch diese muss erst einmal angesteuert werden – daher das `Frame`-Objekt.

Für den Fall, dass eine bereits laufende App neu gestartet wird, prüft die `OnLaunched()`-Methode zunächst, ob das Fenster der App (`Window.Current`) bereits ein `Frame`-Objekt enthält. Falls nicht, wird ein `Frame`-Objekt instanziiert und mit dem Fenster verbunden:

```
if (rootFrame == null)
{
    rootFrame = new Frame();

    // ...

    // Den Rahmen im aktuellen Fenster platzieren
    Window.Current.Content = rootFrame;
}
```

Anschließend wird mithilfe der `Navigate()`-Methode des `Frame`-Objekts die Startseite der App aufgerufen.

```
rootFrame.Navigate(typeof(MainPage), args.Arguments)
```

Zum guten Schluss wird nur noch sichergestellt, dass das Fenster der App auch aktiv und im Vordergrund ist:

```
Window.Current.Activate();
```

## Sonstige Projektdateien

Tabelle 33.3 bietet eine Übersicht über die verbleibenden Dateien eines Windows Store-Projekts.

Datei	Beschreibung
<i>App.xaml</i> und <i>App.xaml.cs</i>	XAML- und C#-Code der Anwendung
<i>MainPage.xaml</i> und <i>MainPage.xaml.cs</i>	XAML- und C#-Code der Seite
<i>Common</i>	Ordner für diverse Hilfsdateien, die von den Projekt- und Seitenvorlagen verwendet werden. Die Projektvorlage <i>Leere App</i> legt hier nur eine einzige Datei ab: <i>StandardStyles.xaml</i> , eine XAML-Datei mit diversen Stilen zur Formatierung der Benutzeroberfläche.
<i>Assets</i>	Ordner für diverse App-Ressourcen. Die Projektvorlagen legen hier z. B. die Bilder für die Kachel und den Begrüßungsbildschirm ( <i>SplashScreen</i> ) ab. ▶



Datei	Beschreibung
<i>Package.appxmanifest</i>	Die Manifestdatei enthält Informationen über die App, die sowohl vom Windows Store als auch dem ausführenden System ausgewertet werden – beispielsweise welche Drehungen die App unterstützt, welche Bilddateien für die Kacheln verwendet werden sollen oder welche besonderen »Funktionen« die App nutzen möchte. Die Manifestdatei ist eine einfache XML-Datei, kann also in jedem XML- oder Texteditor bearbeitet werden. Am sichersten ist es aber, wenn Sie die Manifestdatei mit dem Manifest-Designer von Visual Studio bearbeiten (Doppelklick auf den Knoten der Datei im Projektmappen-Explorer).
<i>APPNAME_TemporaryKey.pfx</i>	Temporäres Zertifikat, das für das Testen und Ausführen der App auf dem lokalen Entwicklungsrechner benötigt wird

**Tabelle 33.3** Projektdateien für Windows Store-Apps

## Startseite austauschen

Die *MainPage*-Seite der Projektvorlage *Leere App* basiert auf der Seitenvorlage *Leere Seite*. Seiten, die mit dieser Seitenvorlage erstellt werden, sind anfangs nicht nur leer, sondern besitzen auch keine weitergehende Unterstützung für die Seitennavigation oder das Sichern und Wiederherstellen von App-Daten.

Für unser aktuelles Beispielprojekt ist dies nicht weiter tragisch, in vielen anderen Fällen werden Sie aber die Startseite durch eine andere Seite austauschen wollen. Gehen Sie dazu wie folgt vor:

4. Löschen Sie die alte Startseite. Klicken Sie dazu im *Projektmappen-Explorer* mit der rechten Maustaste auf den *MainPage.xaml*-Knoten und wählen Sie im Kontextmenü den Befehl *Löschen*.
5. Rufen Sie im Menü *Projekt* den Befehl *Neues Element hinzufügen* auf.
6. Wechseln Sie im erscheinenden Dialogfeld zur Kategorie *Visual C#/Windows Store* und wählen Sie eine Seitenvorlage aus.  
Für eine Seite mit Navigationsstruktur, die ansonsten aber leer ist, gibt es die Vorlage *Standardseite*.  
Die sonstigen Seitenvorlagen sind weiter spezialisiert und besitzen bereits ein vorgegebenes Layout.
7. Ändern Sie gegebenenfalls den Namen der Seite.

Wenn Sie den Namen der Seite in *MainPage* (den Namen der anfänglichen Startseite) ändern, hat dies den Vorzug, dass Sie danach keine Änderungen im Code vornehmen müssen.

Falls Sie dagegen den vorgegebenen Standardnamen beibehalten oder einen eigenen Namen wie z.B. *Startseite* vergeben, müssen Sie noch die Datei *App.xaml.cs* laden und im Aufruf der *Navigate()*-Methode den neuen Seitentyp angeben:

```
// in App.xaml.cs, OnLaunched()-Methode
if (!rootFrame.Navigate(typeof(Startseite), args.Arguments))
{
    throw new Exception("Failed to create initial page");
}
```

8. Klicken Sie auf *Hinzufügen*.

**HINWEIS** Wenn das Projekt bisher nur Seiten enthielt, die auf der Vorlage *Leere Seite* basierten, erscheint vor dem Hinzufügen der Seite noch ein Meldungsfenster, das Ihre Erlaubnis für die Installation weiterer Unterstützungsdateien einholen möchte. Klicken Sie hier auf *Ja*, um die betreffenden Dateien automatisch hinzufügen zu lassen. Die hinzugefügten Dateien finden Sie danach im Unterordner *Common*.

## Schritt 2: Aufbau der Oberfläche

Um etwas Routine im Umgang mit dem Designer zu bekommen, werden wir jetzt ein Image-Steuerelement zum Anzeigen eines Bilds einfügen, sowie ein Label-Element, in das beim Tippen auf das Bild ein Text eingeblendet werden soll (siehe Abbildung 33.4).



**Abbildung 33.4** Die *ErsteApp* im Simulator (nach Antippen des Bilds)

Der Fahrplan für den Aufbau der Benutzeroberfläche sieht wie folgt aus:

- Einrichtung eines zentrierten Layouts basierend auf dem Grid-Element
- Platzierung der Image- und TextBlock-Elemente mithilfe eines StackPanel-Elements
- Aufnahme einer Bildressource und Anzeige im Image-Element

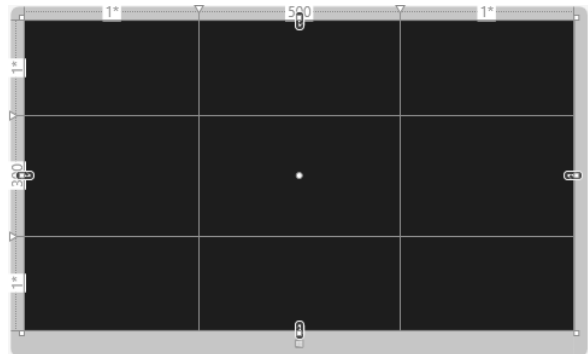
**HINWEIS** Visual Studio verwendet für die Bearbeitung von App-Seiten denselben Designer wie für WPF-Fenster. Wir verzichten daher an dieser Stelle auf eine neuerliche Beschreibung des Designers und verweisen interessierte Leser stattdessen auf den Abschnitt »Der XAML-Designer« in Kapitel 28.

## Zentriertes Layout mit Grid

Selbst App-Seiten, die auf der Vorlage *Leere Seite* basieren, sind nicht gänzlich leer: Sie besitzen bereits ein untergeordnetes Grid-Element. Grid ist ein sehr leistungsfähiger und flexibler Layoutcontainer, der die in ihm eingebetteten Elemente in einer Tabellenstruktur ausrichtet. Anzahl und Breite bzw. Höhe der Spalten und Zeilen dieser Struktur können Sie wahlweise im *Entwurf*-Bereich (bewegen Sie dazu die Maus über die gepunkteten Leisten links und oberhalb des Grids) oder direkt im XAML-Code festlegen.

Für unsere erste App definieren wir eine Tabellenstruktur mit drei Spalten und drei Zeilen. Die Breite der mittleren Spalten setzen wir auf 500 Pixel, die Höhe der mittleren Zeile auf 300 Pixel. Für die Breiten bzw. Höhen der anderen Spalten bzw. Zeilen verwenden wir den Wert »\*«, was bedeutet, dass diese sich den verbliebenen Raum teilen sollen. Auf diese Weise erhalten wir eine mittlere Zelle von 500 mal 300 Pixeln, die zentriert auf der Seite angezeigt wird.

```
<Grid >
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="500" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="300" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
</Grid>
```



**HINWEIS** Wenn Sie den verbliebenen Platz in einem bestimmten Verhältnis zwischen mehreren Zeilen (Spalten) aufteilen möchten, beispielsweise 1:3, setzen Sie die Teilungszahlen vor das Sternsymbol:

```
<Grid.RowDefinitions>
  <RowDefinition Width="1*" />
  <RowDefinition Width="3*" />
</Grid.RowDefinitions>
```

## Steuerelemente einfügen

In die mittlere Zelle sollen nun ein Image-Element zur Anzeige eines Bilds und daneben ein TextBlock-Element zur Anzeige eines Texts eingefügt werden. Statt diese Elemente direkt einzufügen und über feste Margin-Werte zu positionieren, benutzen wir einen StackPanel-Container, der die Elemente automatisch für uns nebeneinander platziert.

Wir fügen also zuerst ein StackPanel-Element in die mittlere Zelle ein. Die gewünschte Zelle wird dabei über ihren Spalten- und Zeilenindex angegeben (Attribute Grid.Column und Grid.Row, wobei zu beachten ist, dass der Index nullbasiert ist).

Damit das StackPanel-Element die Zelle komplett ausfüllt, weisen wir den Attributen HorizontalAlignment und VerticalAlignment den Wert Stretch zu.

Der Wert `Horizontal` für das Attribut `Orientation` teilt dem `StackPanel`-Container mit, dass er die eingebetteten Steuerelemente nebeneinander anzeigen soll.

```
<Grid>
  // ...
  </Grid.RowDefinitions>

  <StackPanel Grid.Column="1" Grid.Row="1"
    HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
    Orientation="Horizontal" >

  </StackPanel>
</Grid>
```

**ACHTUNG** Achten Sie darauf, keine `Margin`-Werte zu setzen, da dies Positionierung und Anordnung verändert.

**HINWEIS** Hinter den Attributen, denen Sie in der XAML-Ansicht Werte zuweisen können, stehen in Realität die `public`-Eigenschaften der entsprechenden Klassen. Statt über den XAML-Code können Sie diese Eigenschaften auch im Eigenschaftenfenster (Aufruf über den Menübefehl *Ansicht/Eigenschaftenfenster*) bearbeiten.

Nun können wir die eigentlichen Steuerelemente in den `StackPanel`-Container einfügen.

```
<Grid>
  // ...
  </Grid.RowDefinitions>

  <StackPanel Grid.Column="1" Grid.Row="1"
    HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
    Orientation="Horizontal" >

    <Image />
    <TextBlock FontSize="40" Text="TextBlock"
      VerticalAlignment="Bottom"/>

  </StackPanel>
</Grid>
```

Das `Image`-Element ist derzeit noch nicht zu sehen, da es mit keiner Bilddatei verknüpft ist (dies erfolgt erst im nächsten Abschnitt). Zuvor fügen wir aber noch in der linken oberen Ecke der Seite ein `TextBlock`-Element ein, in dem wir den Namen der App anzeigen:

```
<Grid>
  // ...
  </Grid.RowDefinitions>

  <TextBlock HorizontalAlignment="Left" VerticalAlignment="Top"
    Margin="50,50,0,0"
    Text="ErsteApp" FontSize="50" Grid.ColumnSpan="2" />

  <StackPanel Grid.Column="1" Grid.Row="1"
    HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
    Orientation="Horizontal" >
```

```
<Image />
<TextBlock FontSize="40" Text="TextBlock"
  VerticalAlignment="Bottom"/>

</StackPanel>
</Grid>
```

Damit das Steuerelement links oben auf der Seite angezeigt wird, müssen wir es in die linke obere Zelle einfügen. Die betreffenden Grid-Attribute, `Grid.Column="0"` und `Grid.Row="0"`, brauchen wir aber nicht explizit anzugeben, da sie den Standardwerten entsprechen.

Außerdem verankern wir das Steuerelement diesmal über die `HorizontalAlignment` bzw. `VerticalAlignment`-Attribute am linken bzw. oberen Zellenrand. Damit das Steuerelement aber nicht direkt am Rand klebt, rücken wir es mit `Margin`-Werten jeweils um 50 Pixel von den Rändern ab. Die Einstellung `Grid.ColumnSpan="2"` sorgt schließlich dafür, dass sich das `TextBlock`-Steuerelement, dessen Breite sich per Voreinstellung nach seinem Inhalt richtet, notfalls über zwei `Grid`-Spalten erstrecken kann. Diese Einstellung ist insofern wichtig, als die Breite der ersten `Grid`-Zeile ja davon abhängt, wie breit der physikalische Anzeigebereich ist. Auf kleinen Displays oder im Snap-Modus kann es schnell passieren, dass die mittlere Zeile den gesamten Anzeigebereich einnimmt und für die erste und dritte Zeile kein zu verteilernder Platz mehr übrig bleibt.

## Bilder in ein Projekt aufnehmen und anzeigen

Zum weiteren Nachvollziehen des Beispielprojekts benötigen Sie zwei Bilder. Am besten nehmen Sie dazu die beiden Bilder *Dogh07.png* und *Dogh09.png*, die Sie im Verzeichnis *Kap33* der Beispielsammlung zu diesem Buch finden. Notfalls können Sie aber auch beliebige andere Bilder verwenden, die allerdings nicht zu groß und möglichst mit transparentem Hintergrund ausgestattet sein sollten.

---

**HINWEIS** Unterstützte Bildformate sind z. B. JPG, PNG und BMP.

Bilder, die für den Aufbau der Benutzeroberfläche benötigt werden, legen Sie am besten im *Assets*-Ordner ab.

9. Klicken Sie dazu im Projektmappen-Explorer mit der rechten Maustaste auf den *Assets*-Ordner und wählen Sie den Befehl *Hinzufügen/Vorhandenes Element*.
10. Wählen Sie die gewünschten Bilder im folgenden Dialogfeld aus und klicken Sie auf *Hinzufügen*.

Die Bilder werden daraufhin in den *Assets*-Ordner des Projektverzeichnisses kopiert und dem Projekt hinzugefügt. Bilder, die dem Projekt hinzugefügt wurden, können Sie bequem über das *Eigenschaftenfenster* mit einem `Image`-Steuerelement verbinden.

11. Wählen Sie zuerst das `Image`-Steuerelement im Designer aus.
12. Blättern Sie dann im *Eigenschaftenfenster* (Aufruf über den Menübefehl *Ansicht/Eigenschaftenfenster*) zur Eigenschaft *Source*<sup>2</sup> und wählen Sie das gewünschte Bild im Dropdown-Listefeld aus.

Sie können das Bild natürlich aber auch direkt über den XAML-Code zuweisen:

---

<sup>2</sup> Am besten lassen Sie die Eigenschaften im *Eigenschaftenfenster* dazu nach Namen und nicht nach Kategorien sortiert anzeigen.

```

<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  // ...

  <TextBlock HorizontalAlignment="Left" VerticalAlignment="Top" Margin="50,50,0,0"
    Text="ErsteApp" FontSize="50" Grid.ColumnSpan="2" />

  <StackPanel Grid.Column="1" Grid.Row="1"
    HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
    Orientation="Horizontal">
    <Image Source="Assets/Dogh07.png" />
    <TextBlock FontSize="40" Text="TextBlock"
      VerticalAlignment="Bottom"/>
  </StackPanel>
</Grid>

```

**Listing 33.5** Aus *MainPage.xaml* (Projekt *ErsteApp*)

### Schritt 3: Ereignisbehandlung

Ebenso wie Desktop-Anwendungen leben Windows Store-Apps davon, dass der Anwender mit ihnen interagiert. Sämtliche Benutzerinteraktionen werden dabei vom Betriebssystem abgefangen und an die App weitergeleitet, wo sie zuerst von den WinRT-Klassen verarbeitet und als Ereignisse weitergemeldet werden. Der App-Entwickler ist damit in der komfortablen Lage, dass er sich in der Regel nur noch überlegen muss, für welches Steuerelement er auf welche Ereignisse reagieren möchte.

Im Falle unserer ersten Beispiel-App möchten wir reagieren, wenn der Benutzer auf das angezeigte Bild klickt: Das angezeigte Bild soll sich ändern, von einem Hundekopf in einen Hundekopf mit geöffnetem Maul, und in dem TextBlock-Steuerelement soll der Text »Wuff« angezeigt werden. Der ernste Hintergrund dieser nicht ganz ernst gemeinten App: Wir wollen sehen, wie man Ereignisse behandelt und wie man in Ereignisbehandlungsmethoden bei Bedarf die Elemente der Benutzeroberfläche manipuliert.

### Vorbereitende Maßnahmen

Zur besseren optischen Kontrolle haben wir in dem TextBlock-Element bereits einen Text angezeigt (*TextBlock*). Dieser soll nun gelöscht werden, damit auf der App-Seite anfangs nur das Bild (mit dem ersten Hundekopf) angezeigt wird.

Entfernen Sie im XAML-Code das `Text`-Attribut aus dem TextBlock-Element:

```

<TextBlock FontSize="40"
  VerticalAlignment="Bottom"/>

```

### Ereignisse mit Behandlungsmethoden verbinden

Wie die Eigenschaften können Sie auch die Ereignisse der Steuerelemente wahlweise im Eigenschaftfenster oder in der XAML-Ansicht bearbeiten (wobei *Bearbeitung* in diesem Fall bedeutet, dass Sie das Ereignis mit einer passenden Behandlungsmethode verbinden).

## Eigenschaftfenster

13. Klicken Sie in der *Entwurf*-Ansicht auf das Element, für das Sie ein Ereignis bearbeiten möchten.
14. Rufen Sie das Eigenschaftfenster auf (Menübefehl *Ansicht/Eigenschaftfenster*).
15. Wechseln Sie im Eigenschaftfenster zur *Ereignisse*-Seite (Blitzsymbol rechts oben).
16. Blättern Sie die Liste der verfügbaren Ereignisse bis zu dem Ereignis, das Sie bearbeiten möchten. Für unsere Beispiel-App wäre dies das Ereignis *Tapped*, das sowohl beim Antippen mit dem Finger als auch beim Anklicken mit der Maus ausgelöst wird.
17. Doppelklicken Sie in das Eingabefeld.

Visual Studio legt daraufhin in der Code-Behind-Datei der aktuellen Seite eine passende Behandlungsmethode an, verbindet diese mit dem Ereignis und lädt den Code der Behandlungsmethode in den Editor, damit Sie den Code eintragen können, der bei Eintritt des Ereignisses ausgeführt werden soll.

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

    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
    }

    private void Image_Tapped_1(object sender, TappedRoutedEventArgs e)
    {
    }
}
```

**HINWEIS** Alle Ereignisbehandlungsmethoden verfügen über zwei Parameter: Der erste Parameter ist immer vom Typ `object` und enthält eine Referenz auf das auslösende UI-Element. Der zweite Parameter dient dazu, weitere Details über das Ereignis an die Behandlungsmethode weiterzureichen und ist oft von einem speziellen Typ, der vom behandelten Ereignis abhängt.


Der folgende Code nutzt z.B. die Informationen, die in der `PointerType`-Eigenschaft des `TappedRoutedEventArgs`-Arguments stecken, um nur auf Fingergesten zu reagieren:

```
private void Image_Tapped_1(object sender, TappedRoutedEventArgs e)
{
    if (e.PointerDeviceType == Windows.Devices.Input.PointerDeviceType.Touch)
    {
        // Hier steht der Behandlungscode, der bei
        // einem Tippen mit dem Finger ausgeführt werden soll
    }
}
```

## XAML-Ansicht

18. Setzen Sie in der XAML-Ansicht die Schreibmarke an eine freie Stelle des XAML-Elements, für das Sie ein Ereignis bearbeiten möchten. Tippen Sie ein Leerzeichen ein.

Im Editor springt ein Popupfenster mit den Eigenschaften und Ereignissen der Komponente auf. Die Ereignisse sind durch ein Blitzsymbol gekennzeichnet.

19. Wählen Sie mithilfe der Pfeiltasten das gewünschte Ereignis aus (für unser Beispiel *Tapped*) und drücken Sie die -Taste.

Es wird ein Attribut für das Ereignis angelegt. Gleichzeitig wird ein Popupfenster zur Auswahl der Ereignisbehandlungsmethode eingeblendet. (Falls Sie das Popupfenster nicht sehen, löschen Sie hinter dem Ereignisnamen die Zeichenfolge "=" und tippen Sie das Gleichheitszeichen erneut ein.)

20. Doppelklicken Sie im Popupfenster auf den Eintrag *<Neuer Ereignishandler>*.

Der Designer legt in der Code-Behind-Datei den Rumpf für die zugehörige Ereignisbehandlungsmethode an und trägt deren Namen als Wert des Ereignisattributs ein.

21. Klicken Sie mit der rechten Maustaste irgendwo in das Attribut und wählen Sie im Kontextmenü den Befehl *Zum Ereignishandler navigieren* aus.

## Zugriff auf XAML-Elemente von C#-Code aus

Um vom C#-Code aus auf Elemente der Benutzeroberfläche zugreifen zu können, die mit XAML definiert wurden, müssen Sie den betreffenden XAML-Elementen Namen zuweisen.

Im Falle unserer Beispiel-App wären dies das Image- und das TextBlock-Element:

```
<StackPanel Grid.Column="1" Grid.Row="1"
    HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
    Orientation="Horizontal">

    <Image Name="dogImage" Source="Assets/Dogh07.png" Tapped="Image_Tapped_1" />
    <TextBlock Name="dogText" FontSize="40" VerticalAlignment="Bottom"/>

</StackPanel>
```

Für XAML-Elemente, denen ein Name zugewiesen wird, legt der Designer im Code der Seite automatisch ein Feld an, das den angegebenen Namen trägt. Über dieses Feld können Sie dann in Ihrem C#-Code auf das betreffende UI-Element zugreifen.

```
using System;
...
using Windows.UI.Xaml.Media.Imaging;

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



```

protected override void OnNavigatedTo(NavigationEventArgs e)
{
}

private void Image_Tapped_1(object sender, TappedRoutedEventArgs e)
{
    // Dem Image-Element dogImage ein anderes Bild zuweisen
    dogImage.Source = new BitmapImage(new Uri(@"ms-appx:/Assets/Dogh09.png"));

    // Dem TextBlock-Element dogText einen anderen Text zuweisen
    dogText.Text = "Wuff";
}
}
}

```

**Listing 33.6** Behandlung des Tapped-Ereignisses (Projekt *ErsteApp*)

Bilder, die Sie im *Assets*-Ordner abgelegt und dem Projekt hinzugefügt haben, können Sie direkt durch Angabe des absoluten Pfads und mithilfe der Klasse `BitmapImage` laden. Das Präfix *ms-appx* in der Pfadangabe steht dabei für das App-Verzeichnis.

**HINWEIS** Die Felder, die der Designer für Ihre XAML-Elemente anlegt, sind standardmäßig `private`. Sie können aber über das XAML-Attribut `x:FieldModifier` einen anderen Zugriffsmodifizierer festlegen:

```
<TextBlock Name="fieldName" x:FieldModifier="public" ... />
```

## Asynchrone Programmierung zur Vermeidung von App-Abstürzen

Der Code, der als Reaktion auf ein Ereignis ausgeführt wird, sollte in wenigen Sekunden abgearbeitet sein, da es ansonsten passieren kann, dass die Laufzeitumgebung die Reaktionsunfähigkeit Ihrer App registriert und die App beendet.

Manchmal lassen sich zeitaufwändige Operationen aber nicht vermeiden: beispielsweise, wenn umfangreiche Dateien zu laden sind oder Daten aus dem Internet angefordert werden. Um auch in solchen Situationen die Benutzeroberfläche reaktionsfähig zu halten und die App vor der Zwangsbeendigung zu bewahren, sollten Sie auf asynchrone Ausführungsmodelle zurückgreifen. C# und die WinRT unterstützen Sie dabei mit den Schlüsselwörtern `async` und `await` (siehe Kapitel 23), sowie zahlreichen Methoden, die asynchron ausgeführt werden und im Namen am Suffix `Async` zu erkennen sind.

So können Sie z.B. größere Bilder oder umfangreiche Bildsammlungen mithilfe der Klasse `StorageFile` und der `BitmapImage`-Methode `SetSourceAsync()` laden:

```

using System;
...
using Windows.Storage;
using Windows.Storage.Streams;
using Windows.UI.Xaml.Media.Imaging;

// ...
private void Image_Tapped_1(object sender, TappedRoutedEventArgs e)

```

```

{
    Uri uri = new Uri(@"ms-appx:/Assets/bild.png");
    StorageFile imgFile = await StorageFile.GetFileFromApplicationUriAsync(uri);

    using (IRandomAccessStream fileStream =
        await imgFile.OpenAsync(Windows.Storage.FileAccessMode.Read))
    {
        BitmapImage bitmapImage = new BitmapImage();

        await bitmapImage.SetSourceAsync(fileStream);
        img1.Source = bitmapImage;
    }
}

```

## Wichtige Ereignisse

Die WinRT-Steuerelemente erben die meisten ihrer Ereignisse von der Klasse UIElement.

Ereignis	Kategorie
Tapped DoubleTapped RightTapped Holding	Finger- und Mausereignisse
DragEnter DragLeave DragOver Drop	Drag & Drop-Ereignisse
GotFocus LostFocus	Fokusereignisse
KeyDown KeyUp	Tastaturereignisse
ManipulationCompleted ManipulationDelta ManipulationInertiaStarting ManipulationStarted ManipulationStarting	Gesten mit mehreren Fingern
PointerCanceled PointerCaptureLost PointerEntered PointerExited PointerMoved PointerPressed PointerReleased PointerWheelChanged	Stift-, Maus- und Fingereingaben

**Tabelle 33.4** Allgemeine UI-Ereignisse

**HINWEIS** Die meisten der in Tabelle 33.4 aufgeführten Ereignisse werden in der Hierarchie der UI-Elemente aufsteigend an die übergeordneten UI-Elemente weitergereicht (Bubbling). Für eine Beschreibung dieses Mechanismus siehe in Kapitel 28 den Abschnitt zu »Bubbling« und »Tunneling«.

## Direkte Überschreibung virtueller Ereignismethoden in abgeleiteten Klassen

Der weiter oben beschriebene Mechanismus der Ereignisbehandlung durch die Verknüpfung von Ereignissen mit Ereignisbehandlungsmethoden ist dazu gedacht, sich via Delegation in den Code fertiger Objekte einzuhaken. Wenn Sie jedoch WinRT-Klassen nicht nur instanziiieren, sondern auch von ihnen ableiten (wie es z.B. für App-Seiten oder bei der Implementierung eigener Steuerelemente üblich ist), können Sie auch direkt in die Ereignisverarbeitung eingreifen, indem Sie die von der Basisklasse definierten virtuellen On-Ereignismethoden überschreiben. Ein typisches Beispiel für diesen Weg der Ereignisbehandlung ist z.B. die OnLaunched()-Methode der App-Seiten (siehe weiter oben Listing 33.4).

## Schritt 4: Kompilieren und ausführen

Zum guten Schluss können Sie das Projekt mit den gewohnten Befehlen erstellen (siehe Menü *Erstellen*) und ausführen (*Start*-Befehle im *Debuggen*-Menü). Bei der ersten Erstellung wird die App auf dem lokalen Rechner installiert und kann danach auch über die Windows-Startseite aufgerufen werden.

**HINWEIS** Apps, die Sie zum Debuggen ausführen, können Sie über den Visual Studio-Befehl *Debugging beenden* beenden. Sie können die App natürlich auch mit der entsprechenden Fingergeste oder durch Drücken der Tastenkombination **[ALT] + [F4]** beenden. Wird die App auf dem lokalen Rechner ausgeführt (statt im Simulator; siehe nächster Abschnitt), dauert es aber in der Regel ein paar Sekunden, bis der Debugmodus wirklich beendet wird.

## Schritt 5: Testen und debuggen

Für das Testen und Debuggen stellt Ihnen Visual Studio noch ein zusätzliches Werkzeug zur Verfügung: den Simulator.

22. Um eine App im Simulator auszuführen, müssen Sie lediglich in der *Standard*-Symbolleiste das Listenelement für die Debugziele aufklappen und den Eintrag *Simulator* auswählen.

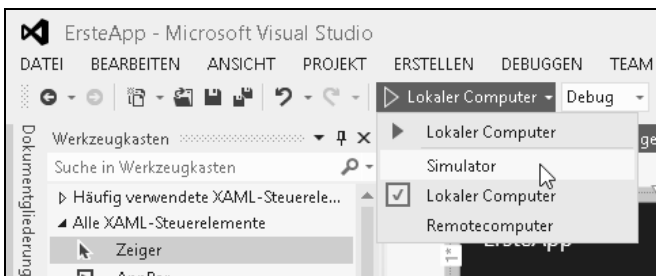


Abbildung 33.5 Aktivierung des Simulators

23. Anschließend führen Sie die App wie gewohnt mit einem der *Start*-Befehle im Menü *Debuggen* aus.



**Abbildung 33.6** Ausführung der App im Simulator

Im Simulator können Sie unter anderem Fingergesten mit der Maus simulieren, zwischen Hoch- und Querformat wechseln und verschiedene Auflösungen einstellen.

---

**HINWEIS** Der Simulator bleibt über die aktuelle Debugsitzung hinaus geöffnet. Er kann über sein Taskleistensymbol beendet werden.

---