Kapitel 2

Anatomie einer Windows Store-App

In diesem Kapitel:

Eine Windows 8 App wird durch die Klasse `Application` repräsentiert. Sie kümmert sich um die Kommunikation mit dem Betriebssystem und die Verwaltung des Lebenszyklus. Zudem ist sie mit einer einzelnen `Window`-Instanz verknüpft, die wiederum für die Darstellung der Oberfläche verantwortlich ist.

Die eigentliche Oberfläche wird hingegen durch `Page`-Objekte repräsentiert. Die Darstellung mehrerer paralleler Fenster ist hierbei nicht möglich. Stattdessen findet eine Navigation zwischen den einzelnen `Page`-Instanzen statt. Diese werden von einem umschließenden `Frame`-Objekt verwaltet, welches sich auch um die Navigation zwischen den Seiten kümmert.

Abbildung 2.1 illustriert die beschriebene Aufteilung.

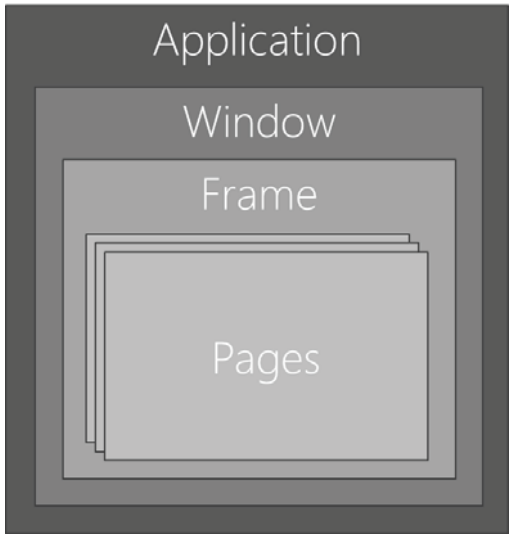

Abbildung 2.1 Die Bestandteile einer Windows-App

Wie Sie sehen, existiert pro App jeweils nur eine `Frame`-Instanz, die mit einem `Window`-Objekt verbunden ist. Dieses übernimmt hierbei die Low-Level-Kommunikation mit dem Betriebssystem und kümmert sich um die Bereitstellung der gesamten Oberfläche.

Schauen Sie sich zum besseren Verständnis einmal das folgende Codefragment aus der Datei *App.cs* an, die in jedem App-Projekt enthalten ist:

```
sealed partial class App : Application
{
  …
  protected override async void OnLaunched(LaunchActivatedEventArgs args)
  {
    var rootFrame = new Frame();
    …
    if (!rootFrame.Navigate(typeof(MainPage)))
    {
      throw new Exception("Failed to create initial page");
    }
    Window.Current.Content = rootFrame;
    Window.Current.Activate();
  }
}
```

Hier wurde zunächst ein neues Frame-Element erzeugt. Die Bereitstellung der Startseite erfolgt nun indirekt durch Navigation. Hierzu nimmt die Methode Navigate() den Typ der entsprechenden Seite entgegen. Das Frame-Objekt kümmert sich nun selbstständig darum, eine entsprechende Instanz zu erstellen und zur Anzeige zu bringen. Funktioniert dies nicht – beispielsweise, weil der angegebene Typ keinem Page-Element entspricht –, gibt sie den Wert false zurück.

Daraufhin wird über Window.Current die aktuelle Fensterinstanz ermittelt und dieser über die Eigenschaft Content das zuvor erstellte Frame-Objekt zugewiesen. Für die Anzeige sorgt schließlich der Aufruf der Activate()-Methode.

Die Rolle der Application-Klasse

Die Klasse Application ist der Dreh- und Angelpunkt einer App. Sie kommuniziert mit dem Betriebssystem und ist für die Verwaltung der App verantwortlich. Dies betrifft zum Beispiel die Verarbeitung von unbehandelten Exceptions. Hierfür bietet die Klasse den Event UnhandledException, der immer dann aufgerufen wird, wenn eine Exception an irgendeiner Stelle der App aufgetreten, aber nicht durch einen try-catch-Block abgefangen wurde. Jede App sollte diesen Event abonnieren und den Benutzer über den aufgetretenen Fehler informieren. Macht sie dies nicht, so beendet Windows einfach die App, ohne dem Benutzer dies zu melden.

Das folgende Beispiel zeigt die notwendigen Anpassungen in der jeweiligen App-Klasse:

```
sealed partial class App : Application
{
  public App()
  {
    this.UnhandledException += this.OnUnhandledException;
    this.InitializeComponent();
  }

  private async void OnUnhandledException(object sender, UnhandledExceptionEventArgs e)
  {
    e.Handled = true;
    await new MessageDialog("Ein interner Fehler ist aufgetreten.\n\n" + e.Message).ShowAsync();
  }
}
```

In der Methode OnUnhandledException() wurde zunächst die Eigenschaft Handled auf true gesetzt. Hiermit signalisieren Sie, dass Sie den Fehler selbst verarbeiten und kein automatisches Herunterfahren der App erforderlich ist. Im Anschluss wird der aufgetretene Fehler mithilfe der Klasse MessageDialog ausgegeben. Hierfür können Sie auf die Message-Eigenschaft des übergebenen UnhandledExceptionEventArgs-Objekts zurückgreifen. Alternativ lässt sich auch die Exception-Eigenschaft auswerten. Diese gibt ein Exception-Objekt zurück, welches neben der Fehlermeldung auch eine interne Fehlernummer (HResult) sowie eine Kette verknüpfter Fehler (InnerException) enthält.

Wenn Sie den Fehler so wie im oberen Beispiel ausgeben, sollten Sie darauf achten, die Eigenschaft Handled stets zuerst zu setzen. Denn durch den Aufruf von asynchronen Methoden (wie der ShowAsync()-Methode von MessageDialog) kann es passieren, dass die App bereits heruntergefahren wird, noch bevor Sie dies durch Setzen der Eigenschaft unterbinden können.

Neben dem beschriebenen Fehlerbehandlungsmechanismus bietet die Klasse Application eine Reihe weiterer Funktionen, auf die später noch im Detail eingegangen wird.

Zugriff auf das Fenster

Die Klasse `Window` repräsentiert den Ankerpunkt der Anwendung. Sie ist nicht nur für die Darstellung der Oberfläche, sondern auch für die Verarbeitung von Eingaben, etwa Tastatur, Maus oder Touch, verantwortlich. Darüber hinaus kümmert sie sich um die verschiedenen Darstellungsformen wie Vollbild oder den Snapped-Modus. Hierbei ist die Funktionalität auf zwei Klassen aufgeteilt: `Window` und `CoreWindow`.

Die Klasse Window

Die `Window`-Klasse hat eine recht einfache Schnittstelle, über die Sie beispielsweise auf Größenänderungen reagieren können.

Die Klasse kann nicht instanziiert werden, sondern wird automatisch vom jeweiligen `Application`-Objekt erstellt. Wenn Sie auf Eigenschaften, Methoden oder Events von `Window` zugreifen wollen, müssen Sie zunächst die aktive Instanz ermitteln. Hierzu stellt die Klasse die statische Eigenschaft `Current` bereit. Um zum Beispiel die Ausmaße des aktiven Fensters zu ermitteln, müssen Sie daher wie folgt vorgehen:

```
Rect windowBounds = Window.Current.Bounds;
```

In der folgenden Tabelle finden Sie die wichtigsten Member der Klasse `Window`:

Member	Beschreibung
`Bounds`	Gibt eine `Rect`-Struktur zurück, welche die Ausmaße des Fensters enthält
`Content`	Enthält den anzuzeigenden Inhalt. Zwar ist die Eigenschaft vom Typ `UIElement`, in der Praxis wird sie jedoch fast immer mit einer `Frame`-Instanz gefüllt.
`CoreWindow`	Gibt die zugehörige `CoreWindow`-Instanz zurück
`Dispatcher`	Bietet Zugriff auf die `Dispatcher`-Klasse, welche für das Management des UI-Threads verantwortlich ist
`Visible`	Dient zur Steuerung der Sichtbarkeit des Fensters
`Activate()`	Bringt das Fenster in den Vordergrund
`Activated`	Wird ausgelöst, wenn die App in den Vordergrund kommt und den Eingabefokus erhält
`SizeChanged`	Wird ausgelöst, wenn sich die Größe des Fensters ändert. Dies kann zum Beispiel beim Wechseln in den Snapped-Modus der Fall sein.
`VisibilityChanged`	Wird ausgelöst, wenn sich die Sichtbarkeit des Fensters ändert

Tabelle 2.1 Die wichtigsten Member der Klasse `Window`

Die Klasse CoreWindow

`CoreWindow` bietet hingegen Zugriff auf Low-Level-Funktionalität – zum Beispiel für das Abfangen von Tastaturereignissen oder das Abfragen von Systemtasten. So können Sie beispielsweise die Events `KeyUp`, `KeyDown`, `PointerPressed` und `PointerReleased` verwenden, um auf Tastenanschläge oder Mausklicks zu reagieren. Zudem lässt sich über die Methode `GetKeyState()` explizit abfragen, ob eine bestimmte Taste gedrückt wurde. Die Eigenschaften `PointerPosition` und `PointerCursor` liefern hingegen die aktuelle Position beziehungsweise das Erscheinungsbild des Mauszeigers.

Der folgende Code setzt zum Beispiel den Mauszeiger der App in den Status `Wait`:

```
Window.Current.CoreWindow.PointerCursor = new CoreCursor(CoreCursorType.Wait, 0);
```

Navigation mit der Klasse Frame steuern

Die Klasse `Frame` übernimmt nicht nur die Bereitstellung von Seiten, sondern kümmert sich auch um die Navigation zwischen diesen. Hierfür stellt sie unter anderem die vorhin genannte Methode `Navigate()` zur Verfügung. Diese nimmt ein Objekt vom Typ `TypeName` entgegen, welches die entsprechende Seite beschreibt. Der folgende Code zeigt hierfür ein Beispiel:

```
rootFrame.Navigate(typeof(MainPage));
```

Optional können Sie auch eine Überladung von `Navigate()` verwenden, die zusätzlich einen Initialisierungsparameter entgegennimmt:

```
rootFrame.Navigate(typeof(DetailPage), myData);
```

Die `Navigate()`-Methode gibt einen booleschen Wert zurück, der den Erfolg der Navigation signalisiert. Diesen sollten Sie in jedem Fall auswerten, da im Falle eines Fehlers nicht automatisch eine Exception ausgelöst wird. Wie Sie später sehen werden, kann es auch vorkommen, dass die Zielseite die Navigation abbricht.

BEGLEITDATEIEN Die Begleitdateien zu den folgenden Beispielen finden Sie im Verzeichnis:

```
...\Kapitel 2\NavigationDemo
```

Da die Navigation jedoch von den Seiten selbst veranlasst wird, müssen diese die Aufgabe an die übergeordnete `Frame`-Instanz delegieren. Hierfür enthält jedes `Page`-Objekt die Eigenschaft `Frame`. Der Wechsel von einer Seite zu einer anderen könnte somit wie folgt aussehen:

```
this.Frame.Navigate(typeof(DetailPage), myData);
```

Bei jedem Zugriff auf die `Frame`-Eigenschaft sollten Sie prüfen, ob diese bereits gefüllt wurde. Denn erst nach der vollständigen Beendigung des Navigationsvorgangs wird die `Frame`-Eigenschaft der Page mit einer jeweiligen Instanz belegt. Aus diesem Grund können beispielsweise Zugriffe auf die `Frame`-Eigenschaft im Konstruktor der Zielseite zu einem Fehler führen, da zum Zeitpunkt der Erstellung noch kein Frame zugeordnet wurde.

Aus diesem Grund sollte der obere Code besser wie folgt umgeschrieben werden:

```
if (this.Frame != null)
  this.Frame.Navigate(typeof(DetailPage), myData);
```

Jede Navigation wird vom jeweiligen Frame-Objekt protokolliert. Daher muss sich die jeweilige Page-Instanz nicht selbst merken, von welcher Seite zu ihr navigiert wurde. Denn eine Kenntnis darüber ist notwendig, wenn der Benutzer zurück beziehungsweise vorwärts navigieren möchte.

Auch für diese Art der Navigation bietet die Klasse Frame entsprechende Methoden. So navigiert GoBack() beispielsweise auf die vorherige Seite, während GoForward() eine Vorwärtsnavigation durchführt. Ob eine solche Navigation im aktuellen Kontext möglich ist, kann über die Eigenschaften CanGoBack und CanGoForward abgefragt werden. Diese sollten Sie stets vor der Navigation abfragen, wie das folgende Beispiel zeigt:

```
if (this.Frame != null && this.Frame.CanGoBack)
  this.Frame.GoBack();
```

Übergabeparameter auf der Zielseite auswerten

Wenn Sie bei der Navigation einen Parameter übergeben, sollten Sie diesen auf der Zielseite auswerten. Hierfür können Sie zum Beispiel den Event Navigated abonnieren. Er liefert nicht nur die Übergabeparameter, sondern auch den Typ der Seite, von der navigiert wurde. Der folgende Code zeigt ein Beispiel:

```
public sealed partial class DetailPage : Page
{
  public DetailPage()
  {
    this.InitializeComponent();
    this.Loaded += OnLoaded;
  }

  private void OnLoaded(object sender, RoutedEventArgs e)
  {
    if (this.Frame != null)
      this.Frame.Navigated += Frame_Navigated;
  }

  void Frame_Navigated(object sender, NavigationEventArgs e)
  {
    MyData data = e.Parameter as MyData;
  }
}
```

Der Navigated-Event wurde hierbei erst nach dem vollständigen Laden der Seite abonniert. Hintergrund dieser Maßnahme ist, dass die hierfür erforderliche Frame-Instanz erst zu diesem Zeitpunkt verfügbar ist.

Beim Eintreten des Events können die Übergabedaten ausgelesen werden. Diese stellt das übergebene NavigationEventArgs-Objekt über die Parameter-Eigenschaft bereit. Mithilfe der Eigenschaft SourcePageType können Sie darüber hinaus den Typ der Seite ermitteln, von der navigiert wurde.

Ob zum ersten Mal auf die entsprechende Seite navigiert wurde, signalisiert die Eigenschaft NavigationMode der NavigationEventArgs-Klasse. Dies ist beispielsweise der Fall, wenn sie den Wert New zurückgibt. Diese Informationen können Sie zum Beispiel verwenden, um eine Initialisierung der Seite vorzunehmen, die nur beim ersten Öffnen ablaufen soll:

```
private void Frame_Navigated(object sender, NavigationEventArgs e)
{
    if (e.NavigationMode == NavigationMode.New)
        // Initialisierung...
}
```

Gibt `NavigationMode` hingegen den Wert `Refresh` zurück, signalisiert dies, dass zur aktuellen `Page`-Instanz bereits navigiert wurde und nun gegebenenfalls lediglich andere Daten übergeben wurden. Die Werte `Back` oder `Forward` zeigen hingegen an, dass eine Rückwärts- oder Vorwärtsnavigation zu einer existierenden `Page`-Instanz vollzogen wurde.

Navigation abbrechen

Wenn Sie auf der Zielseite feststellen, dass diese in einem invaliden Zustand ist, oder ein Wechsel zu der Seite aus irgendeinem anderen Grund nicht sinnvoll erscheint, können Sie die Navigation auch abbrechen. Hierfür bietet die Klasse `Frame` den `Navigating`-Event. Dieser liefert ein Objekt vom Typ `NavigatingCancelEventArgs`. Setzen Sie dessen `Cancel`-Eigenschaft auf `true`, unterbricht dies die Navigation und der Benutzer bleibt auf der Ursprungsseite.

```
this.Frame.Navigating += this.Frame_Navigating;
…
private void Frame_Navigating(object sender, NavigatingCancelEventArgs e)
{
  …
  e.Cancel = true;
}
```

In diesem Fall sollte auch die Quellseite über die fehlgeschlagene Navigation informiert werden. `Frame` bietet hierfür die Events `NavigationStopped` und `NavigationFailed`. Während Ersterer darüber informiert, dass die Navigation von der Zielseite abgebrochen wurde, signalisiert Letzterer eine fehlgeschlagene Navigation.

Historie ermitteln und zuweisen

Wie weiter oben bereits erwähnt, protokolliert die Klasse `Frame` den Verlauf der Navigation. Dies ist nicht nur nützlich, um eine leichte Vorwärts- und Rückwärtsnavigation zu ermöglichen, sondern auch zur Speicherung des Zustands. Denn eine App sollte ihren Zustand, inklusive der zuletzt angezeigten Seite, speichern, bevor diese von Windows angehalten wird.

Hierfür bietet `Frame` die Methoden `GetNavigationState()` und `SetNavigationState()`. Während Erstere eine Zeichenkette mit dem Verlauf der Navigation zurückgibt, stellt Letztere den gespeicherten Zustand wieder her. Das folgende Beispiel zeigt, an welchen Stellen der Anwendung Sie das Speichern und Wiederherstellen der Navigationshistorie hinterlegen müssen:

```
sealed partial class App : Application
{
  public App()
  {
    this.InitializeComponent();
```

```
    this.Suspending += OnSuspending;
  }

  protected override async void OnLaunched(LaunchActivatedEventArgs args)
  {
    Frame rootFrame = Window.Current.Content as Frame;
    if (rootFrame == null)
    {
      rootFrame = new Frame();
      if (args.PreviousExecutionState == ApplicationExecutionState.Terminated)
      {
        // Navigationshistorie ermitteln und zuweisen
        var state = ... // gespeicherte Historie aus Datei ermitteln
        rootFrame.SetNavigationState(state);
      }
      Window.Current.Content = rootFrame;
    }
    …
  }

  private async void OnSuspending(object sender, SuspendingEventArgs e)
  {
    var deferral = e.SuspendingOperation.GetDeferral();

    // Navigationshistorie speichern
    var frame = Window.Current.Content as Frame;
    var state = frame.GetNavigationState();
    // Historie speichern
    // ...

    deferral.Complete();
  }
}
```

HINWEIS Im Beispielprojekt zu diesem Abschnitt finden Sie eine vollständige Implementierung, inklusive des Speicherns und Ladens der Navigationshistorie.

Wenn Sie die Visual Studio-Projektvorlagen *Raster-App* oder *Geteilte App* verwenden, brauchen Sie sich nicht explizit um das Laden und Speichern der Historie zu kümmern, da dies die mitgelieferte Basisklasse `LayoutAwarePage` beziehungsweise die Klasse `SuspensionManager` automatisch übernimmt.

Standardmäßig speichert die Klasse `Frame` die Namen der letzten zehn Seiten der Navigationshistorie. Wenn Sie diesen Wert verändern wollen, können Sie die Eigenschaft `CacheSize` entsprechend anpassen.

Navigation auf Seitenebene

Wie Sie gesehen haben, finden fast alle Navigationsoperationen über die Klasse `Frame` statt. Die Klasse `Page` bietet jedoch einige Hilfsmittel, die den Umgang erleichtern. So verfügt sie beispielsweise über die virtuelle Methode `OnNavigatedTo()`, mit der dediziert auf den Navigationsprozess Einfluss genommen werden kann. Sie wird ausgeführt, wenn auf die aktuelle Seite navigiert wird.

Dies ist besonders dann nützlich, wenn Sie Übergabeparameter der Quellseite auswerten müssen. Zudem entfällt das manuelle Abonnieren des `Frame.Navigated`-Events. Stattdessen brauchen Sie lediglich die Methode `OnNavigatedTo()` zu überschreiben, wie das folgende Beispiel demonstriert:

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

  protected override void OnNavigatedTo(NavigationEventArgs e)
  {
    var data = e.Parameter as string;
  }
  …
}
```

Darüber hinaus bietet die `Page`-Klasse mit den virtuellen Methoden `OnNavigatingFrom()` und `OnNavigatedFrom()` die Möglichkeit zu handeln, bevor auf eine andere Seite navigiert wird. Dies ist nicht nur der Fall, wenn Sie explizit auf eine andere Seite navigieren, sondern auch bei der Vorwärts- und Rückwärtsnavigation.

Beide Methoden können vor allem dann nützlich sein, wenn Sie die Navigation von bestimmten Voraussetzungen abhängig machen wollen. Hierbei ist es sinnvoll, die Voraussetzungen an einer zentralen Stelle zu prüfen und nicht vor jedem Navigationsvorgang. Denn in der Praxis gibt es neben dem expliziten Navigieren über `Navigate()` auch stets die Rückwärtsnavigation. Um nun nicht an zwei Stellen redundanten Code zur Prüfung der Voraussetzungen zu hinterlegen, sollten Sie stattdessen die Methode `OnNavigatingFrom()` überschreiben. Der folgende Code prüft beispielsweise, ob auf der aktuellen Seite Validierungsfehler vorliegen. Wenn dem so ist, wird die Navigation abgebrochen:

```
protected override void OnNavigatingFrom(NavigatingCancelEventArgs e)
{
    if (this.ValidateData() == false)
        e.Cancel = true;
    base.OnNavigatingFrom(e);
}
```

`OnNavigatedFrom()` eignet sich hingegen als zentraler Ankerpunkt für alle Operationen, die vor dem Verlassen der Seite stattfinden müssen. Das Speichern der gerade bearbeiteten Daten wäre hierfür ein Beispiel:

```
protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    this.SaveData();
    base.OnNavigatedFrom(e);
}
```

Page Caching

Bei der Navigation auf eine Seite wird standardmäßig eine neue Instanz erstellt, unabhängig davon, ob der Benutzer zuvor auf dieser unterwegs war und lediglich eine Rückwärtsnavigation vollzogen hat.

Dieses Verhalten können Sie jedoch ändern. Die Klasse `Page` bietet hierfür die Eigenschaft `NavigationCacheMode`, die auf einen der folgenden Werte gesetzt werden kann:

- `Disabled` Es erfolgt keine Zwischenspeicherung der Seite, sondern stets eine Neuerstellung. Dies ist der Standardwert von `NavigationCacheMode`.
- `Enabled` Wurde die Seite bereits angesprungen, wird die Instanz wiederverwendet. Es sei denn, sie liegt außerhalb der maximalen Cachegröße, die auf Ebene der zugehörigen `Frame`-Instanz festgelegt wurde (`CacheSize`-Eigenschaft).
- `Required` Die Seite wird in jedem Fall wiederverwendet, unabhängig davon, ob sie sich außerhalb der Cachegröße befindet

Bevor Sie die `NavigationCacheMode`-Eigenschaft ändern, sollten Sie jedoch die Vor- und Nachteile gegeneinander abwägen. Seiten, die aus dem Cache kommen, sind sehr performant, da keine vollständige Neuerstellung der Oberfläche, inklusive der Daten, notwendig ist. Dies macht sich insbesondere dann bemerkbar, wenn die anzuzeigenden Daten erst über einen externen Webservice ermittelt werden müssen.

Auf der anderen Seite steigt jedoch auch der Speicherverbrauch, da die Seiten nicht sofort entsorgt werden. Dies kann nun dazu führen, dass Windows Ihre App schneller terminiert, da hierbei der Speicherverbrauch eine entscheidende Rolle spielt. Auch wenn Sie hierbei den State Ihrer App speichern und der Benutzer nach dem Neustart dort weiterarbeiten kann, wo er aufgehört hat, bleibt die initiale Startzeit.

Dieses Austarieren zwischen Performance und Ressourcenverbrauch können Sie jedoch sehr feingranular steuern. Zum einen lässt sich die `NavigationCacheMode`-Eigenschaft für jede Seite separat festlegen. Hierbei lässt sich abwägen, wie viele Ressourcen die Seite beansprucht beziehungsweise wie viel Zeit die Initialisierung der Seite benötigt. Über die `CacheSize`-Eigenschaft der `Frame`-Klasse besteht zudem die Möglichkeit, einzustellen, wie viele Seiten maximal zwischengespeichert werden sollen.

Zum anderen können Sie den gesamten Speicherbedarf Ihrer App sowie die Terminierung durch Windows sehr gut im Windows Task-Manager überwachen.

Das Laufzeitverhalten einer App

Windows Store-Apps haben ein grundlegend anderes Laufzeitverhalten als klassische Anwendungen. Während Desktop-Anwendungen so lange laufen, bis der Benutzer sie beendet, werden WinRT-Apps automatisch heruntergefahren. Grundlage hierfür sind verschiedene Faktoren, beispielsweise der verbrauchte Arbeitsspeicher oder die letzte Nutzung durch den Benutzer. Als einfache Regel gilt: Je weniger Arbeitsspeicher Ihre App verbraucht, desto länger lebt sie. Darüber hinaus hält das Betriebssystem Apps an, die zurzeit nicht vom Benutzer in Verwendung sind. Hierbei bleibt die App zwar im Arbeitsspeicher, verbraucht jedoch keine Prozessorzeit oder Netzwerkverbindungen. Dieser Vorgang wird *Suspension* genannt. Er verringert den Ressourcenverbrauch, was sich positiv auf die Akkulaufzeit und die Performance der Apps auswirkt. Abbildung 2.2 illustriert das Lebenszeitmodell von Apps in Windows 8.

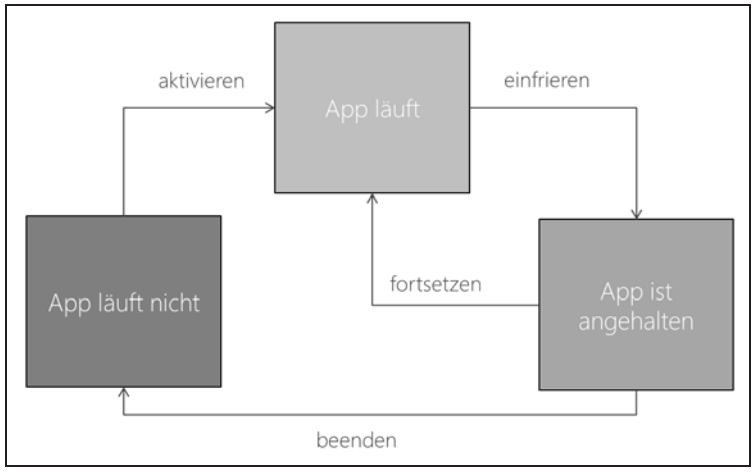

Abbildung 2.2 Der Lebenszyklus einer App in Windows 8

Um ein Gefühl für die Funktionsweise des Suspension-Vorgangs zu bekommen, sollten Sie einige Apps öffnen, zwischen diesen springen und sich die Aktivitäten im Windows Task-Manager anschauen. In Abbildung 2.3 sehen Sie, dass zwar mehrere Apps ausgeführt werden, diese jedoch weder Prozessor- noch Disk-, noch Netzwerkoperationen durchführen, da sie im Suspension-Modus laufen.

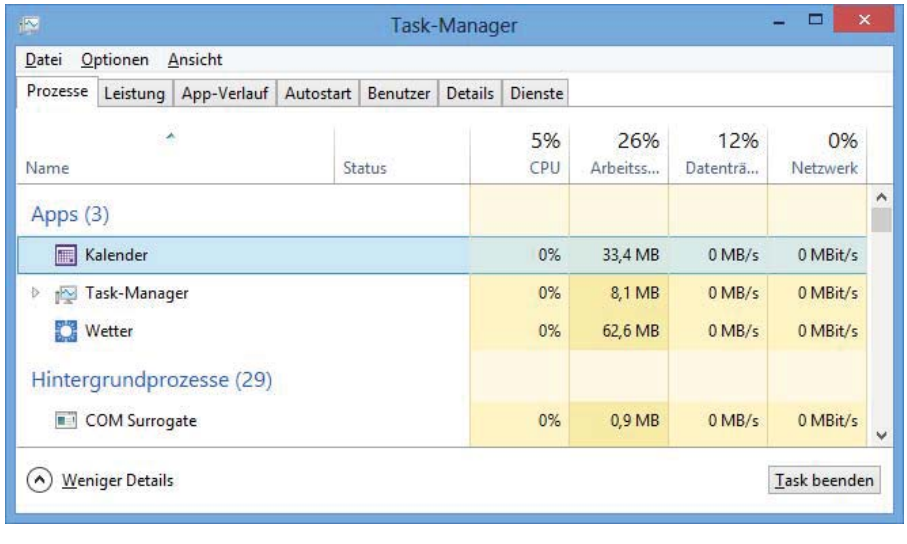

Abbildung 2.3 Der Task-Manager zeigt an, dass inaktive Apps keine Prozessor-, Disk- oder Netzwerkoperationen durchführen

Darüber hinaus werden Sie im Task-Manager feststellen, dass einzelne Apps nach einiger Zeit vollständig beendet werden.

Auf den Suspension-Modus reagieren

Auf den Suspension-Automatismus müssen Sie sich als App-Entwickler einstellen. Denn das Betriebssystem geht nicht nur davon aus, dass jede App in einer bestimmten Zeit ihre Daten speichert, wenn sie angehalten wird, sondern schreibt der App auch vor, wie sie sich nach einer erneuten Aktivierung verhalten soll. Die Entwicklungsrichtlinien sehen hierbei vor, dass die App vor dem Anhalten nicht nur die aktuellen Daten, sondern auch die aktuelle Position des Benutzers innerhalb der App speichert. Auf diese Weise bemerkt der Benutzer nicht, dass die App zwischendurch heruntergefahren wurde und kann genau da weiter machen, wo er zuletzt aufgehört hat. Obwohl das Speichern der Position beim Einfrieren nicht nötig ist, da sich die App ja noch im Arbeitsspeicher befindet, sollten Sie zu diesem Zeitpunkt immer eine Speicherung vorsehen, da die App zu einem späteren Zeitpunkt automatisch heruntergefahren werden kann.

Damit Ihre App mitbekommt, wenn ein bestimmter Status eintritt, informiert Windows sie in Form verschiedener Ereignisse. Die folgende Tabelle listet diese Ereignisse und beschreibt deren Auftreten:

Ereignis	Von	Zu
`Activated`	App läuft nicht	App läuft
`Suspending`	App läuft	App ist eingefroren
`Resuming`	App ist eingefroren	App läuft

Tabelle 2.2 Die Ereignisse des App-Lebenszyklus und der Zeitpunkt ihres Auftretens

Speichern des aktuellen Zustands

Um mitzubekommen, wenn ein bestimmtes Ereignis auftritt, können Sie zum Beispiel den `Suspending`-Event der Klasse `Application` abonnieren. Das folgende Listing zeigt ein Beispiel:

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

  private void OnSuspending(object sender, SuspendingEventArgs e)
  {
    var deferral = e.SuspendingOperation.GetDeferral();

    // TODO: Hier den Status der letzten Session ermitteln

    deferral.Complete();
  }
}
```

Bevor Sie jedoch mit dem Speichern beginnen, sollten Sie zunächst einen so genannten *Deferral* ermitteln. Dies ist erforderlich, da die meisten WinRT-APIs, wie zum Beispiel Dateisystemoperationen, asynchron ausgeführt werden. In diesem Fall würde jedoch die Eventhandler-Methode beendet, bevor die eigentliche Operation ausgeführt wurde. Das `Deferral`-Objekt sorgt nun dafür, dass der Mainthread auf die Beendigung der parallelen Operationen wartet, bis die Methode `Complete()` ausgeführt wurde.

Zudem sollten Sie den Speichervorgang so effizient wie möglich gestalten. Denn nach dem Auftreten des `Suspending`-Events hat die App genau 5 Sekunden Zeit, ihre Daten zu speichern. Dauert dies länger, beendet sie Windows vollständig.

Zudem sollten Sie alles speichern, was für ein vollständiges Wiederherstellen des aktuellen Zustands nötig ist, auch wenn der `Suspending`-Event lediglich das Anhalten der App signalisiert. Denn wenn Windows Ihre App später terminiert, bekommt Ihre App hiervon nichts mit.

Status der App in Visual Studio steuern

Wenn Sie eine App aus Visual Studio heraus starten, wird der Suspension-Event unterdrückt, damit Sie die Ausführung mit dem Debugger überwachen können. Um jedoch die fehlerfreie Ausführung des Suspension-Codes testen zu können, bietet Visual Studio eine entsprechende Funktion. Um sie zu aktivieren, müssen Sie zunächst die Symbolleiste *Debugspeicherort* (oder *Debug Location* in der englischen Version) einblenden (Menü: *Ansicht/Symbolleisten/Debugspeicherort*). Daraufhin können Sie die App über ein Symbol anhalten oder beenden.

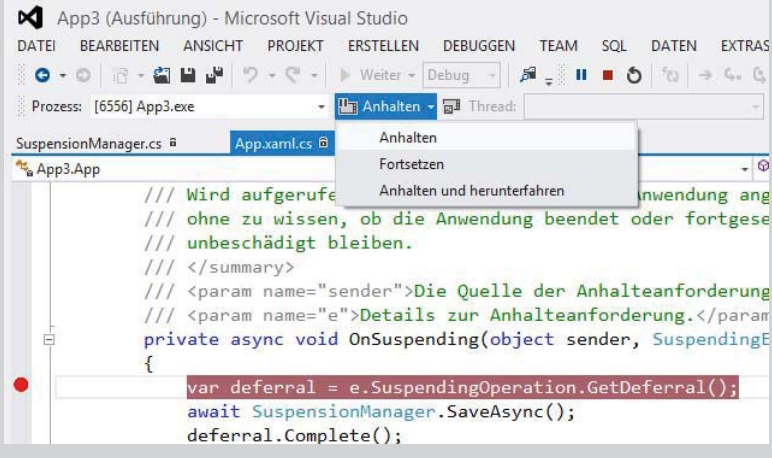

Abbildung 2.4 Anhalten, Fortsetzen oder Beenden einer App über den Debugger veranlassen

Wiederherstellen des gespeicherten Zustands

Für die Aktivierung der App – ob nun aus dem angehaltenen Zustand oder beim Neustart – müssen Sie keinen Event konsumieren, sondern können stattdessen die Methode `OnLaunched()` überschreiben. Diese bekommt als Parameter ein Objekt vom Typ `LaunchActivatedEventArgs` übergeben, welches den vorherigen Ausführungsstatus der App über die Eigenschaft `PreviousExecutionState` signalisiert. Das folgende Listing zeigt dies an einem einfachen Beispiel:

```
sealed partial class App : Application
{
  …
  protected override void OnLaunched(LaunchActivatedEventArgs args)
  {
    if (args.PreviousExecutionState == ApplicationExecutionState.Running)
```

```
    {
      Window.Current.Activate(); // App läuft bereits, lediglich aktivieren
      return;
    }

    // App instanziieren
    // …

    if (args.PreviousExecutionState == ApplicationExecutionState.Terminated)
    {
      // TODO: Hier den Status der letzten Session ermitteln
    }

    // App starten
    …
  }
}
```

HINWEIS Neben dem Abfragen des Status in der Methode `OnLaunched()` können Sie auch den `Resuming`-Event abonnieren, um unmittelbar über das Fortsetzen der App informiert zu werden.

Hier wurde zunächst geprüft, ob die App zuvor ausgeführt (und aus dem *Suspended*-Status kommt) oder zuvor vollständig beendet wurde. Im ersten Fall müssen Sie keine Zustandsdaten ermitteln, da sich die App noch vollständig im Arbeitsspeicher befindet und nun lediglich aktiviert werden muss. In diesem Fall reicht es, das entsprechende Fenster in den Vordergrund zu bringen. Wurde die App zuvor vollständig beendet, so sollten Sie den letzten Zustand wieder herstellen.

Wenn Sie nur einfache Zustandsdaten speichern und laden, müssen Sie dies nicht vollständig selbst implementieren, sondern können auf die Klasse `SuspensionManager` zurückgreifen. Diese wird automatisch generiert, wenn Sie die Visual Studio-Vorlagen *Raster-App* oder *Geteilte App* zur Projektanlage verwenden.

Das folgende Listing enthält die öffentlichen Eigenschaften und Methoden der Klasse mit einer kurzen Beschreibung:

```
internal sealed class SuspensionManager
{
  // Dictionary für die Verwaltung von Zustandswerten.
  public static Dictionary<string, object> SessionState { get; }

 // Liste von benutzerdefinierten Typen für die Serialisierung.
  public static List<Type> KnownTypes { get; }

  // Speichert die Daten.
  public static async Task SaveAsync();

  // Ermittelt die gespeicherten Daten.
  public static async Task RestoreAsync();

  // Registriert ein Frame zur automatischen Speicherung der Navigationshistorie.
  public static void RegisterFrame(Frame frame, String sessionStateKey);

  // Deregistriert ein Frame von der automatischen Speicherung der Navigationshistorie.
  public static void UnregisterFrame(Frame frame);
```

```
  // Dient zur Speicherung von Frame-spezifischen Daten.
  public static Dictionary<String, Object> SessionStateForFrame(Frame frame);
}
```

Mithilfe von `SuspensionManager` reduziert sich das Speichern und Laden von Werten auf wenige Zeilen Code. In den folgenden Zeilen wird beispielsweise in einer fiktiven Shop-App die ID der aktuell angezeigten Bestellung gespeichert, um sie nach dem erneuten Öffnen der App wieder anzuzeigen:

```
// Schreiben:
SuspensionManager.SessionState["CurrentOrderID"] = _currentOrderId;
await SuspensionManager.SaveAsync();

// Lesen:
await SuspensionManager.RestoreAsync();
_currentOrderId = (int)SuspensionManager.SessionState["CurrentOrderId"];
```

Die Methode `SessionStateForFrame()` ist für die Verwaltung von seitenspezifischen Einstellungen zuständig und wird von der ebenfalls generierten Basisklasse `LayoutAwarePage` verwendet. Anstatt die Methode direkt zu verwenden, sollten Sie lieber die von `LayoutAwarePage` bereitgestellten Methoden `LoadState()` und `SaveState()` überschreiben und das hierbei übergebene Dictionary entsprechend erweitern.

Capabilities

Alle WinRT-Apps laufen in einer Sandbox. Diese isoliert die Apps voneinander und verhindert ebenfalls den direkten Zugriff auf Systemressourcen. In vielen Fällen möchte der Benutzer jedoch zum Beispiel auf das Internet, seine lokalen Bilder oder Kontakte aus der App zugreifen.

Um dies zu ermöglichen, bietet WinRT das Konzept der *Capabilities*, zu Deutsch *Fähigkeiten*, in Visual Studio etwas unglücklich mit *Funktionen* übersetzt. Hierbei signalisiert die App, welche Systemressourcen benötigt werden, und der Benutzer kann dann selbst entscheiden, ob dieser Zugriff gewährt wird. Hierzu werden entsprechende Deklarationen im Application Manifest der App hinterlegt. Dieses finden Sie in Ihrem Visual Studio-Projekt in Form der Datei *Package.appxmanifest*.

HINWEIS Beim Zertifizierungsprozess, der bei der Bereitstellung einer App im Windows Store durchlaufen werden muss, findet eine Prüfung statt, welche die deklarierten Capabilities mit dem angegebenen Zweck der App abgleicht. Deklariert die App mehr, als sie eigentlich benötigt, wird diese automatisch abgelehnt.

Capabilities konfigurieren

Die Capabilities können sehr einfach in Visual Studio über die Projekteigenschaften verwaltet werden, wie Abbildung 2.5 zeigt.

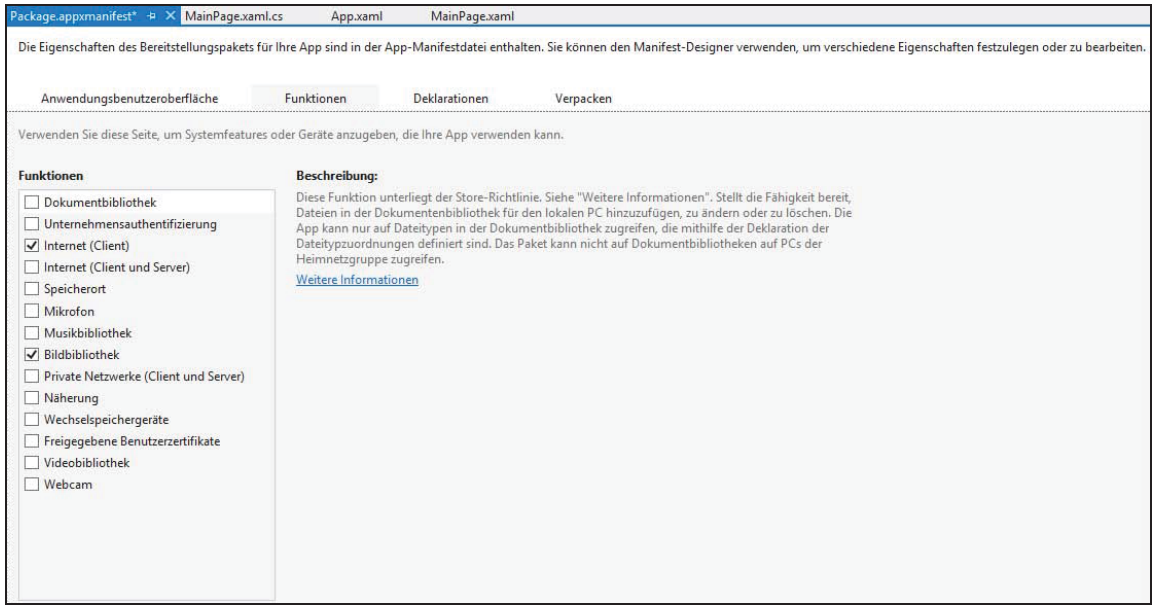

Abbildung 2.5 Pflege der Capabilities in Visual Studio (Registerkarte *Funktionen*)

Capabilities (in der deutschen Visual Studio-Version *Funktionen* genannt) sind nur für den indirekten Zugriff auf Ressourcen erforderlich, bei dem der Benutzer nicht direkt involviert ist. Wenn dieser dagegen beispielsweise über einen Dateidialog ein Bild auswählt, ist hierfür keine Capability notwendig. Um bei diesem Beispiel zu bleiben: Nur wenn die App ohne das Zutun des Benutzers direkt auf das Dateisystem zugreift, wird eine entsprechende Deklaration benötigt.

Die zu Verfügung stehenden Capabilities werden von der Windows Runtime definiert und zur Laufzeit durch den Runtime Broker geprüft. Nachfolgend finden Sie die Liste der zur Verfügung stehenden Capabilities mit einer kurzen Beschreibung:

- **Dokumentenbibliothek** Ermöglicht das Ändern, Löschen und Anlegen von Dateien innerhalb der Dokumentenbibliothek des lokalen Computers. Hierbei muss der Typ der entsprechenden Dateien separat deklariert werden. Hiervon ausgenommen sind Dateien, die über die Netzwerkgruppe *Heimnetzgruppe* verfügbar gemacht wurden. Auf diese kann nur über eine spezialisierte Capability, etwa *Musikbibliothek,* zugegriffen werden.
- **Unternehmensauthentifizierung** Gewährt Zugriff auf Unternehmensressourcen, die eine Anmeldung an eine Domäne erfordern
- **Internet (Client und Server)** Ermöglicht den Zugriff auf ein- und ausgehenden Internetverkehr
- **Internet (Client)** Bietet lediglich ausgehenden Zugriff auf das Internet, zum Beispiel den Zugriff auf einen externen Web Service
- **Speicherort (Location)** Gewährt Zugriff auf die aktuelle Position, die zum Beispiel über GPS, das Drahtlosnetzwerk oder die IP-Adresse ermittelt werden kann
- **Mikrofon** Ermöglicht den Zugriff auf das lokale Mikrofon, um zum Beispiel eine Audioaufnahme zu starten

- **Musikbibliothek** Ermöglicht das Hinzufügen, Ändern oder Löschen von Dateien in der lokalen Musikbibliothek sowie in den Musikbibliotheken der Computer, die sich in der Netzwerkgruppe *Heimnetzgruppe* befinden
- **Bildbibliothek** Ermöglicht das Hinzufügen, Ändern oder Löschen von Dateien in der lokalen Bildbibliothek sowie in den Bildbibliotheken der Computer, die sich in der Netzwerkgruppe *Heimnetzgruppe* befinden
- **Private Netzwerke (Client und Server)** Bietet Zugriff auf den ein- und ausgehenden Verkehr in Heim- oder Firmennetzwerken
- **Näherung** Ermöglicht den Zugriff auf Geräte in der unmittelbaren Nähe, welche über Near Field Communication (NFC) verbunden werden können
- **Wechselspeichergeräte** Bietet die Möglichkeit, Dateien auf Wechseldatenträgern zu ändern, zu löschen oder hinzuzufügen. Hierbei ist jedoch der Typ der Dateien separat zu deklarieren. Ausgeschlossen hiervon sind Dateien, die über die Netzwerkgruppe *Heimnetzgruppe* verfügbar sind.
- **Freigegebene Benutzerzertifikate** Bietet die Möglichkeit, auf Software- und Hardwarezertifikate, inklusive Smart Card-Zertifikate, zuzugreifen
- **Videobibliothek** Ermöglicht das Hinzufügen, Ändern oder Löschen von Dateien in der lokalen Videobibliothek sowie in den Videobibliotheken der Computer, die sich in der Netzwerkgruppe *Heimnetzgruppe* befinden
- **Webcam** Bietet Zugriff auf die lokale Videokamera des Geräts

Bei einigen Capabilities erfolgt beim Zugriff eine separate Benutzerabfrage durch den Runtime Broker. Dies ist beispielsweise bei der Standortlokalisierung oder bei Verwendung der Kamera der Fall.

BEGLEITDATEIEN Die Begleitdateien zu den folgenden Beispielen finden Sie im Verzeichnis:

```
...\Kapitel 2\CapabilitiesDemo
```

Capabilities in einer App verwenden

Nach der notwendigen Theorie geht es nun um die praktische Anwendung. Hierbei soll eine kleine App die aktuelle Position (den Standort) bestimmen. Hierbei wird die Capability *Speicherort* (eine etwas unglückliche Übersetzung des Begriffs *Location*) im App Manifest deklariert, die eine Lokalisierung über einen eingebauten GPS-Chip, den angeschlossenen WLAN-Router oder die lokale IP-Adresse ermöglicht.

Wie Sie in folgendem Code sehen, kommt für die Ermittlung die Klasse `Geolocator` zum Einsatz, die über entsprechende Eigenschaften die Position im Längen- und Breitengrad angibt. Zudem wird versucht, die zughörige postalische Adresse (Postleitzahl, Ort, Land) sowie die lokale Uhrzeit zu ermitteln.

```
var locator = new Geolocator();
var position = await locator.GetGeopositionAsync();

var location = string.Format("Deine Koordinaten sind: {0} Longitude / {1} Latitude\n",
  position.Coordinate.Longitude, position.Coordinate.Latitude);
if (string.IsNullOrWhiteSpace(position.CivicAddress.City))
```

```
{
  location += "Die lokale Adresse ist zurzeit nicht verfügbar.\n";
}
else
{
  location += string.Format("Die Adresse ist: {0}, {1} {2}\n",
    position.CivicAddress.PostalCode, position.CivicAddress.City,
    position.CivicAddress.State, position.CivicAddress.Country);
}
location += string.Format("Die aktuelle Uhrzeit am Standort ist: {0}",
  position.CivicAddress.Timestamp.DateTime.ToString());
```

Wenn Sie diesen Code ausführen, wird zunächst einmal eine `UnauthorizedAccessException` ausgelöst, da der Runtime Broker beim Zugriff auf die Methode `GetGeopositionAsync()` prüft, ob die Capability *Speicherort* (*Location*) im App Manifest deklariert wurde (Abbildung 2.6).

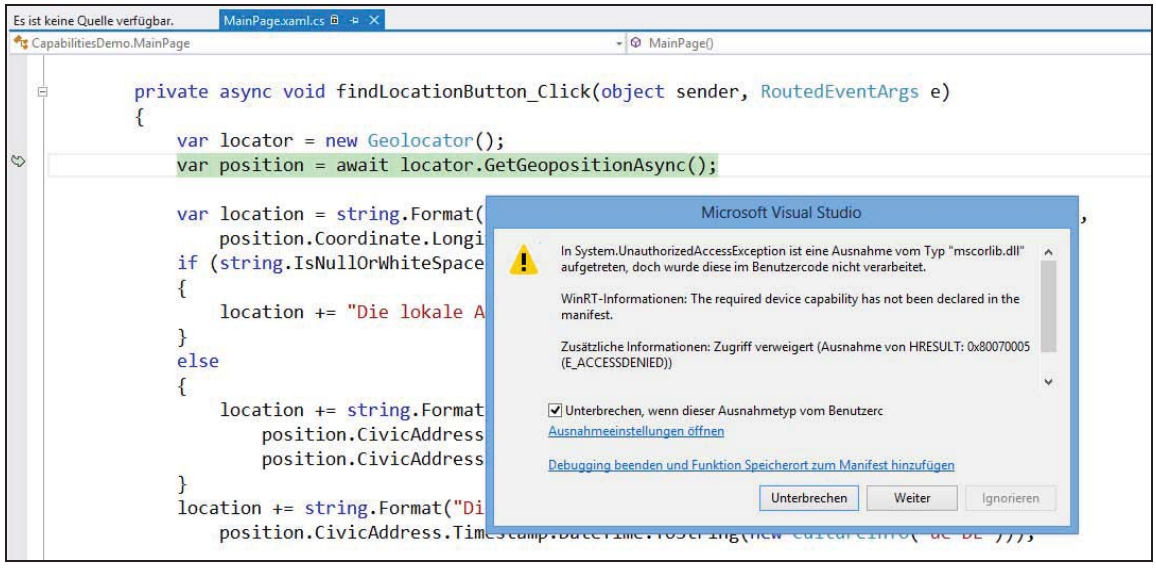

Abbildung 2.6 Beim Start ohne deklarierte Capability kommt es zu einer `UnauthorizedAccessException`

Wenn Sie nun die entsprechende Capability im App Manifest deklarieren und das Programm erneut starten, bekommen Sie beim ersten Zugriff auf die `Geolocator`-Klasse die Sicherheitsabfrage aus Abbildung 2.7.

Abbildung 2.7 Der Runtime Broker fragt vor dem ersten Zugriff beim Benutzer nach

Dieser Mechanismus stellt sicher, dass nicht nur die App eine Capability deklariert hat, sondern der Benutzer ebenfalls in die Entscheidung mit eingebunden wird. Stimmt der Benutzer zu, wird dies in den Einstellungen der App protokolliert und beim nächsten Zugriff nicht mehr gefragt.

App Contracts

App Contracts bieten der App die Möglichkeit, bestimmte Dienste anderer Anwendungen zu nutzen, beziehungsweise selbst Dienste anzubieten. Ein Beispiel für eine solche, anwendungsübergreifende Funktionalität ist die Windows-Suche. Wenn Sie über die Seitenleiste (*Charms Bar*) die Suche aufrufen, liefert sie Resultate aus verschiedenen Quellen, sprich Apps. Wenn Sie nun Ihre eigene App in die Suche einbeziehen wollen, weil sie selbst bestimmte Daten zu Verfügung stellt, die für die Suche relevant sein könnten, so müssen Sie lediglich den entsprechenden Contract unterstützen.

Um die Unterstützung für einen Contract zu realisieren, muss Ihre App eine entsprechende Deklaration im *Application Manifest* (Datei *Package.appxmanifest*) enthalten. Diese Deklarationen können Sie sehr leicht in Visual Studio über die Projekteigenschaften pflegen. Abbildung 2.8 zeigt dies an einem Beispiel.

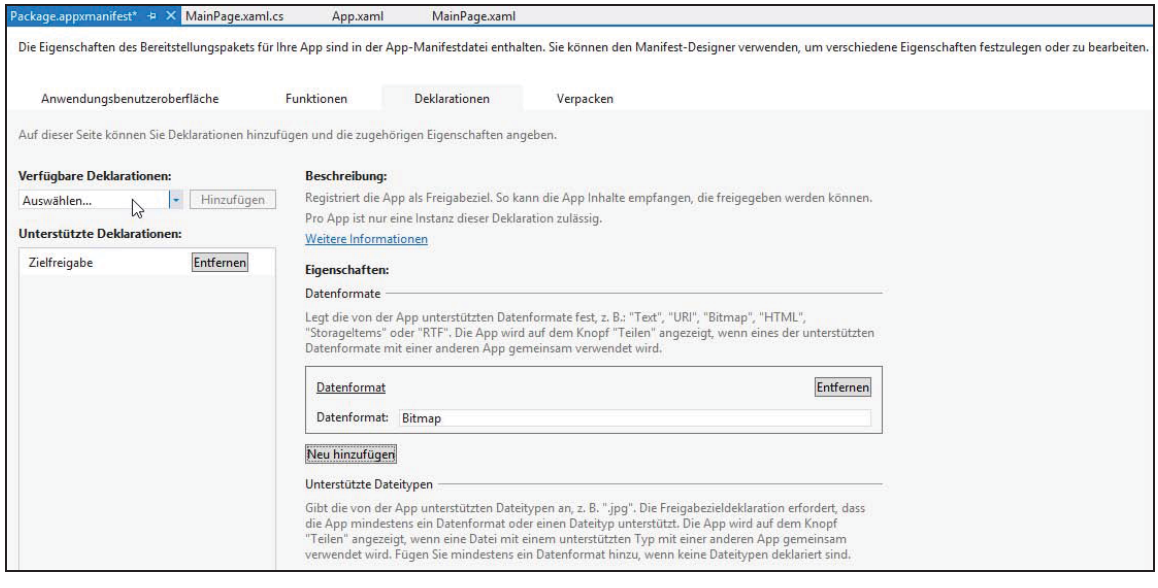

Abbildung 2.8 Die Verwaltung von Deklarationen in den Visual Studio-Projekteigenschaften

Nachfolgend finden Sie eine Liste der zur Verfügung stehenden Deklarationen, inklusive einer kurzen Erläuterung:

- **Aktualisierung zwischengespeicherter Dateien** Erlaubt die automatische Aktualisierung von bestimmten Dateien, die von der App bereitgestellt werden
- **Dateiöffnungsauswahl** Bietet die Möglichkeit, Dateien anderer Apps zu öffnen
- **Dateispeicherungsauswahl** Ermöglicht das Speichern von Inhalten im Dateisystem

- **Dateitypzuordnungen** Bietet den Zugriff auf bestimmte Dateitypen. Auf diese Weise kann eine App beispielsweise eine ausgewählte Datei mit der entsprechend verknüpften App öffnen. Ist für den angegebenen Dateityp keine App registriert, fragt das System den Benutzer.
- **Einstellungen für Druckaufgaben** Hierdurch kann eine App eine eigene Oberfläche für die Druckerkonfiguration bereitstellen
- **Gerät automatisch wiedergeben** Ermöglicht das Öffnen der App, wenn ein bestimmtes Gerät angeschlossen wird
- **Hintergrundaufgaben** Bietet der App die Möglichkeit, bestimmte Aufgaben im Hintergrund zu verrichten, nachdem die eigentliche App bereits heruntergefahren wurde
- **Inhalt automatisch wiedergeben** Bietet der App die Möglichkeit, sich für einen bestimmten Dateityp zu registrieren. Hierdurch wird die App automatisch geladen, wenn eine Datei dieses Typs zum Beispiel über den Windows-Explorer geöffnet wird.
- **Inhaltsauswahl** Ermöglicht der App, auf die Kontakte des aktuellen Benutzers zuzugreifen
- **Kameraeinstellungen** Bietet die Möglichkeit, Kameraoptionen aus der App heraus zu bearbeiten
- **Kontobildanbieter** Ermöglicht es dem Benutzer, sein Profilbild aus der App heraus zu ändern
- **Protokoll** Ermöglicht die Nutzung bestehender Protokolle oder die Erstellung von benutzerdefinierten Protokollen
- **Spiel-Explorer** Ermöglicht der App, sich in Windows als Spiel zu registrieren
- **Suchen** Ermöglicht es der App, in anderen Apps zu suchen, beziehungsweise die Daten der eigenen App in die allgemeine Suche zu integrieren
- **Zertifikate** Erlaubt das Installieren von digitalen Zertifikaten durch die App
- **Zielfreigabe (Share)** Die App fungiert als Ziel einer Teilen (Share)-Operation. Hierbei registriert sie sich für einen bestimmten Dateityp. Hierdurch kann ein Inhalt aus einer anderen App durch diese App geteilt werden. Beispielsweise registriert sich die Mail-App als *Zielfreigabe*, um es anderen Apps zu ermöglichen, über die Charms Bar Inhalte per E-Mail zu verschicken, ohne die entsprechende Funktionalität selbst implementieren zu müssen.

Die App als Quelle eines Contracts

Nach der grauen Theorie soll die Entwicklung von Contract-Unterstützung nun demonstriert werden. Als Beispiel dient der *Zielfreigabe* (*Share*)-Contract. Er ermöglicht das Teilen von Informationen aus einer App heraus. Wenn Ihre App zum Beispiel mit Bildern hantiert, kann es sinnvoll sein, dem Benutzer die Möglichkeit zu geben, ein Bild über die Charms Bar zum Beispiel auf Facebook zu posten. Ein anderes Beispiel wären Texte, die Sie aus der App heraus per E-Mail verschicken möchten.

BEGLEITDATEIEN Die Begleitdateien zu den folgenden Beispielen finden Sie im Verzeichnis:

```
...\Kapitel 2\ContractsDemo
```

Dreh- und Angelpunkt für das Teilen von Informationen ist die Klasse `DataTransferManager` aus dem Namespace `Windows.ApplicationModel.DataTransfer`. Sie ermöglicht unter anderem das Teilen von Texten, Bildern, Links und Dateien. Um die Klasse verwenden zu können, müssen Sie über die statische Methode `GetForCurrentView()` zunächst eine Instanz für Ihre App erzeugen:

```
var dataTransferManager = DataTransferManager.GetForCurrentView();
```

Die Klasse hat eine Verbindung zur Charms Bar und wird automatisch benachrichtigt, wenn der Benutzer in dieser den *Teilen*-Button anwählt. Ihre App meldet sich nun wiederum über den Event `DataRequested` beim `DataTransferManager` an, um die entsprechenden Daten bei Bedarf bereitzustellen. Dies passiert idealerweise in dem Moment, in dem der Benutzer auf die Seite navigiert, sprich in der Methode `OnNavigatedTo()`. In der Methode `OnNavigatedFrom()`, die beim Verlassen aufgerufen wird, sollten Sie den Eventhandler wieder abmelden. Das folgende Listing demonstriert diesen Vorgang:

```
public sealed partial class MainPage : Page
{
  private DataTransferManager _dataTransferManager;

  protected override void OnNavigatedTo(NavigationEventArgs e)
  {
    // Diese Seite als "Share"-Source registrieren
    _dataTransferManager = DataTransferManager.GetForCurrentView();
    _dataTransferManager.DataRequested += this.OnDataRequested;
  }

  protected override void OnNavigatedFrom(NavigationEventArgs e)
  {
    // Diese Seite als "Share"-Source abmelden
    _dataTransferManager.DataRequested -= this.OnDataRequested;
  }

  private void OnDataRequested(DataTransferManager sender, DataRequestedEventArgs e)
  {
    …
  }
}
```

Tritt das `DataRequested`-Ereignis ein, bekommt der Eventhandler die entsprechende `DataTransferManager`-Instanz sowie ein `DataRequestEventArgs`-Objekt als Parameter übergeben. Letzeres bietet über die `Request`-Eigenschaft Zugriff auf die jeweilige Anfrage. Dieses Objekt müssen Sie nun lediglich um die entsprechenden Daten anreichern. Um beispielsweise einen Text bereitzustellen, ist der folgende Code notwendig:

```
private void OnDataRequested(DataTransferManager sender, DataRequestedEventArgs e)
{
  DataPackage requestData = e.Request.Data;
  requestData.Properties.Title = "Titel";
  requestData.Properties.Description = "Beschreibung";
  requestData.SetText("Inhalt");
}
```

Hierbei wurde zunächst ein `DataPackage`-Objekt ermittelt, welches über die `Properties`-Eigenschaft die Zuweisung von Titel und Beschreibung bietet. Beide Informationen werden zur Anzeige in der Charms Bar verwendet und signalisieren dem Benutzer, welche Informationen geteilt werden. Während das Setzen der `Title`-Eigenschaft zwingend erforderlich ist, kann die `Description`-Eigenschaft optional gefüllt werden.

Abbildung 2.9 zeigt die Darstellung der Charms Bar, nachdem der obere Code ausgeführt wurde.

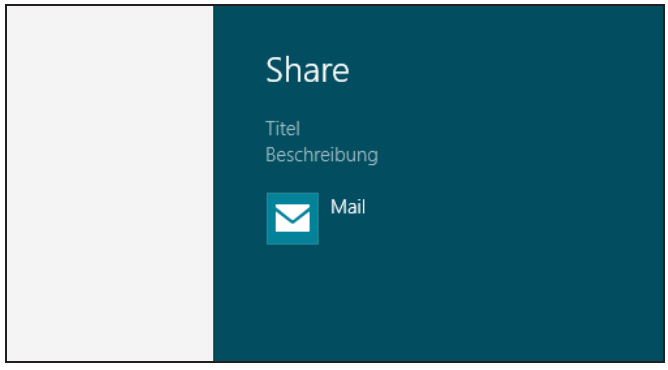

Abbildung 2.9 Über das Teilen-Feature der Charms Bar kann die App einen Inhalt mit anderen Apps teilen

Hier werden dem Benutzer nun alle Apps aufgelistet, die sich für das Teilen von Text registriert haben. In diesem Fall ist das die Mail-App. Nach der Auswahl öffnet sich diese im Snapped-Modus, sodass Sie die E-Mail parallel zu Ihrer App bearbeiten können (Abbildung 2.10).

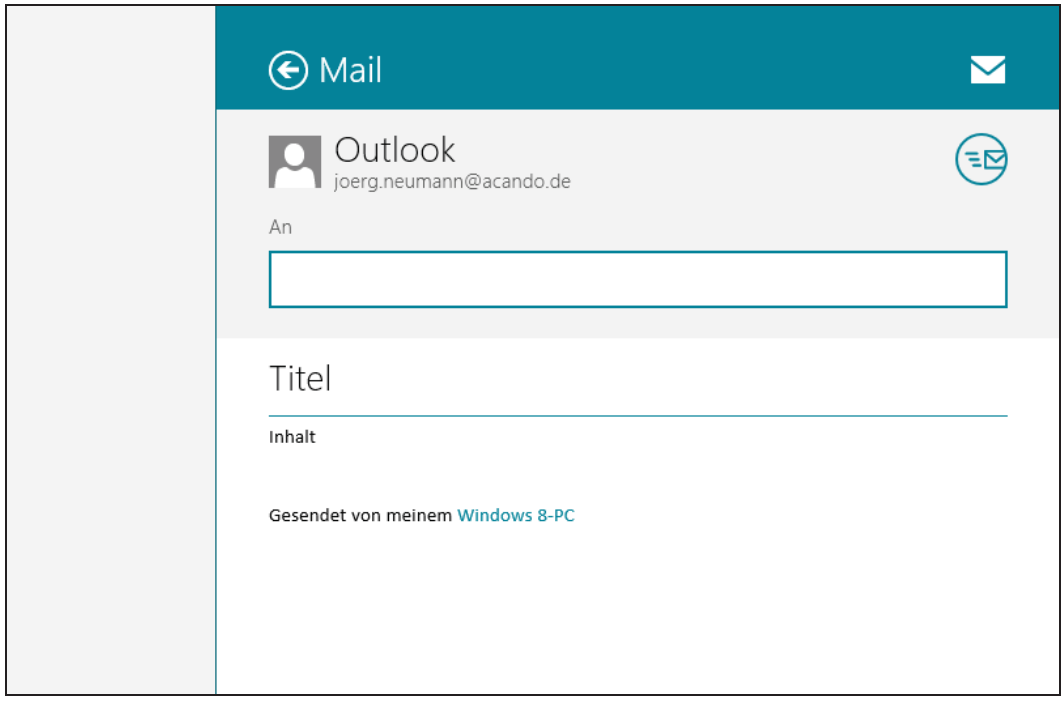

Abbildung 2.10 Die Mail-App blendet sich parallel zur laufenden App ein

Sie können die Charms Bar auch programmtechnisch einblenden. Dies kann zum Beispiel sinnvoll sein, wenn Sie das Teilen einer Information über einen Button anbieten wollen. Hierfür bietet `DataTransferManager` die statische Methode `ShowShareUI()`:

```
DataTransferManager.ShowShareUI();
```

Die App als Ziel eines Contracts

Nachdem Sie gesehen haben, wie Ihre App als Quelle für einen Contract fungieren kann, geht es nun um den umgekehrten Weg. Hierbei bietet Ihre App sich als Ziel für den *Suchen*-Contract an. Dies bedeutet zunächst nur, dass sie in der Suchleiste erscheint. Wählt der Benutzer Ihre App an, wird sie automatisch gestartet. Noch während des Tippens werden daraufhin bis zu fünf Suchergebnisse der App direkt in der Suchleiste angezeigt. Durch Anwahl eines Ergebnisses oder durch das Bestätigen der Suche wird der Suchbegriff an die App übertragen. Diese kann daraufhin auf die eigene Suchseite navigieren und dem Benutzer die Ergebnisse der App-internen Suche präsentieren. Abbildung 2.11 illustriert den Ablauf.

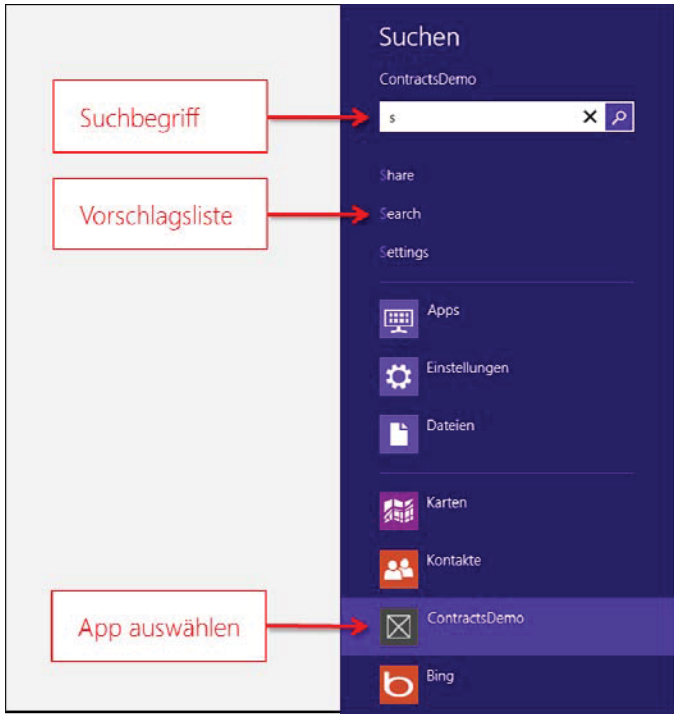

Abbildung 2.11 Über die Suchleiste kann die App gestartet werden

Um Ihre App mit der Suche zu verknüpfen, muss sie zunächst einmal über die nötige Berechtigung verfügen. Hierzu öffnen Sie die App-Manifestdatei (*Package.appxmanifest*) und wechseln auf die Registerkarte *Deklarationen*. Dort wählen Sie aus der Liste der Deklarationen den Punkt *Suchen* aus, wie Abbildung 2.12 zeigt.

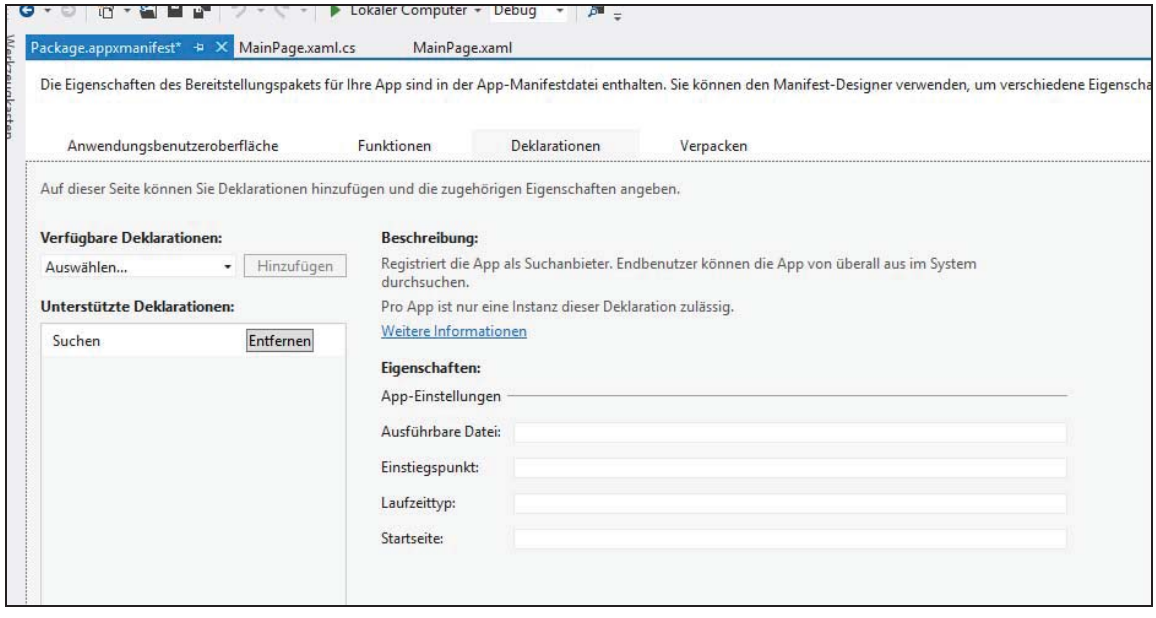

Abbildung 2.12 Konfigurieren der Suchfunktionalität in der App-Manifestdatei

Wenn Sie die Suche in Ihre App einbauen, kann der Benutzer jederzeit über die Charms Bar in Ihrer App nach einem bestimmten Begriff suchen. Hierbei ist es unerheblich, ob Ihre App gerade aktiv ist oder nicht. Ist sie nicht aktiv, startet Windows sie und übergibt hierbei den Suchbegriff.

Dies müssen Sie natürlich in Ihrer App entsprechend berücksichtigen. Die Klasse `Application` bietet hierfür die virtuelle Methode `OnSearchActivated()`. Diese müssen Sie überschreiben und zunächst einmal die App starten. Da hierbei derselbe Code erforderlich ist, den Sie auch in `OnLaunched()` ausführen, ergibt es zunächst einmal Sinn, die Initialisierung der App in einer separaten Methode zu hinterlegen und diese aus `OnLaunched()` und `OnSearchActivated()` heraus aufzurufen.

`OnSearchActivated()` enthält einen Parameter vom Typ `SearchActivatedEventArgs`, der über die Eigenschaft `QueryText` Zugriff auf den jeweiligen Suchtext ermöglicht. Diesen Text sollten Sie an die entsprechende Suchseite weiterleiten und diese im Anschluss öffnen.

Das folgende Beispiel zeigt die notwendigen Änderungen der `App`-Klasse:

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

  protected async override void OnLaunched(LaunchActivatedEventArgs args)
  {
    await EnsureMainPageActivatedAsync(args);
  }
```

```
  protected async override void OnSearchActivated(SearchActivatedEventArgs args)
  {
    await EnsureMainPageActivatedAsync(args);
    MainPage.Current.DataContext = args.QueryText;
  }

  private async Task EnsureMainPageActivatedAsync(IActivatedEventArgs args)
  {
    …
    var rootFrame = new Frame();
    rootFrame.Navigate(typeof(MainPage));
    Window.Current.Content = rootFrame;
    Window.Current.Activate();
  }
  …
}
```

In `OnSearchActivated()` wurde die übergebene Suchanfrage der `DataContext`-Eigenschaft der Hauptseite zugewiesen, welche wiederum für die Bindung der Suchanfrage an die Oberfläche zuständig ist. Der Zugriff auf die aktive Instanz der Seite wurde hierbei über die selbst definierte Eigenschaft `Current` realisiert. Dies gewährleistet, dass die Suchanfrage auch an die Instanz übertragen wird, die beim Starten erzeugt wird.

Die Implementierung dieses Mechanismus sieht wie folgt aus:

```
public sealed partial class MainPage : Page
{
  public static MainPage Current;

  public MainPage()
  {
    this.InitializeComponent();
    Current = this;
  }
  …
}
```

Damit ist die eigentliche Arbeit schon getan. Ihre Suchseite kann nun noch ein zusätzliches Feature namens *Suggestions* unterstützen. Hierbei wird der gerade getippte Text direkt an die Suchseite Ihrer App übermittelt und von dieser analysiert. Die hierbei ermittelten Ergebnisse werden daraufhin in der Suchleiste in Form einer interaktiven Vorschlagsliste angezeigt.

Um die Verbindung mit der Suchleiste herzustellen, müssen Sie sich zunächst eine Instanz der Klasse `SearchPane` ermitteln. Hierfür bietet diese die statische Methode `GetForCurrentView()` an. Um nun auf das Tippen des Suchbegriffs reagieren zu können, sollten Sie den Event `OnSuggestionsRequested` abonnieren.

Das folgende Listing zeigt den grundlegenden Aufbau der Suchseite:

```
public sealed partial class MainPage : Page
{
  private SearchPane searchPane;
  public static MainPage Current;
```

```
  public MainPage()
  {
    this.InitializeComponent();
    Current = this;
  }

  protected override void OnNavigatedTo(NavigationEventArgs e)
  {
    // Diese Seite als "Search"-Target registrieren
    searchPane = SearchPane.GetForCurrentView();
    searchPane.SuggestionsRequested += this.OnSearchPaneSuggestionsRequested;
  }

  protected override void OnNavigatedFrom(NavigationEventArgs e)
  {
    // Diese Seite als "Search"-Target abmelden
    searchPane.SuggestionsRequested -= this.OnSearchPaneSuggestionsRequested;
  }
  …
}
```

Die eigentliche Logik steckt nun in der Eventhandler-Methode `OnSearchPaneSuggestionsRequested()`. Diese bekommt über einen Parameter den jeweiligen Suchtext sowie eine Ergebnisliste übergeben, die mit den entsprechenden Treffern gefüllt werden kann. Das folgende Listing definiert als Beispiel ein Array von Begriffen, das durchsucht wird:

```
public sealed partial class MainPage : Page
{
  private string[] _words = { "Charms", "Share", "Search", "Devices", "Settings" };
  …

  void OnSearchPaneSuggestionsRequested(SearchPane sender, SearchPaneSuggestionsRequestedEventArgs e)
  {
    var queryText = e.QueryText;

    var request = e.Request;
    foreach (string word in _words)
    {
      if (word.StartsWith(queryText, StringComparison.CurrentCultureIgnoreCase))
      {
        // Treffer im Suchbereich anzeigen
        request.SearchSuggestionCollection.AppendQuerySuggestion(word);

        // Bei max. 5 Ergebnissen abbrechen
        if (request.SearchSuggestionCollection.Size >= 5)
          break;
      }
    }
    this.DataContext = queryText;
  }
}
```