Optimierung für verschiedene Ansichten

Windows 8 ist ein Betriebssystem für viele Gerätesorten. Entsprechend macht Microsoft selbstbewusst Werbung damit, dass das Betriebssystem alle Geräteklassen zwischen dem klassischen Desktop-Computer und dem neuesten Tablet abdeckt. Schon allein dieser Umstand sorgt dafür, dass man sich als Entwickler auf eine Vielzahl an Auflösungen und Formfaktoren einstellen muss.

Hinzu kommt, dass Windows 8 selbst unterschiedliche Ansichtsmodi für Apps unterstützt. So sind Apps etwa gezwungen, den *Snap View*-Modus zu unterstützen, also die Ansicht, bei der eine App 320 Pixel Breite des Bildschirms einnimmt, während der übrige Teil des Bildschirms von einer anderen App genutzt werden kann. Darüber hinaus weisen immer mehr tragbare Geräte spezielle Sensoren auf, die registrieren, wenn der Benutzer das Tablet dreht. Der Bildschirminhalt dreht sich dann mit, er schaltet vom Landscape-Modus in den Portrait-Modus um und umgekehrt. Demzufolge muss auch die App in der Lage sein, die momentane Ausrichtung des Bildschirms zu berücksichtigen.

Webentwickler generell sind keineswegs schockiert, wenn man von ihnen die Unterstützung von unterschiedlichen Auflösungen fordert. Denn kaum ein anderer Entwicklungsbereich kann so sehr von sich behaupten, geräteunabhängig entwickeln zu können. Das bedeutet aber nicht, dass die zur Verfügung stehenden Werkzeuge immer ein angenehmes Entwickeln ermöglichen. Windows 8 schließt mit APIs und Werkzeugen einige der Lücken, die sonst Schmerzen bereiten.

In diesem Kapitel werden mehrere Themenbereiche besprochen. Als Erstes wird die von Windows 8 unterstützte Pixelskalierung beleuchtet, mit der unabhängig vom DPI-Wert des jeweiligen Bildschirms entwickelt werden kann. Sie erfahren in diesem Zusammenhang auch, wie Sie sich grundsätzlich von Auflösungsproblematiken befreien können. Im Anschluss daran wird der Snap View betrachtet. Dabei wird darauf eingegangen, wie programmtechnisch und visuell auf dessen Auftreten reagiert werden kann. Anschließend bespricht dieses Kapitel die Geräteorientierung und welche Möglichkeiten Ihnen mit HTML5 und JavaScript zur Verfügung stehen, dieses Feature zu unterstützen. Zu guter Letzt geht es um den Semantic Zoom, also die Möglichkeit, dem Nutzer schnell zu mehr Übersicht zu verhelfen.

8.1 Skalierung und unterschiedliche Auflösungen

Eine grundsätzliche Überlegung in puncto Bildschirmauflösung ergibt, dass bereits ohne Sonderfälle eine ganze Reihe an Auflösungen abgedeckt werden müssen: Absolutes Minimum für Windows 8 sind 1024 × 768 Pixel, wohingegen die minimale Auflösung für »Windows 8 Devices« 1366 × 768 Pixel beträgt. Die höchste Auflösung, die momentan bei typischen Windows 8-Geräten beobachtet werden kann, ist jedoch eine deutlich größere: 2560 × 1440 Pixel. Als wäre das nicht schon genug, werden die Auflösungen auf unterschiedlichste Bildschirmgrößen verteilt: Die hohen 2560 × 1440 Pixel gibt es beispielsweise als Massen-Display bereits mit einer Bildschirmdiagonale von 11,6 Zoll, aber eben auch für größere Displays mit bis zu 27 Zoll. Wer jetzt schon stöhnt, hat möglicherweise noch gar nicht daran gedacht, dass bei Touchgeräten spontan die Hälfte des Bildschirms durch die Bildschirmtastatur belegt werden kann.

Flüssige Layouts als Standard

Microsoft empfiehlt für die Entwicklung grundsätzlich, flüssige Layouts einzusetzen, um unabhängig von einer konkreten Auflösung zu entwickeln. Der antiquierte Ansatz, verschiedene Layouts für unterschiedliche, spezifische Auflösungen zu definieren, ist aufgrund der großen Bandbreite bei DPI-Werten ohnehin ein vergeblicher Kampf, wird aber auch aufgrund der großen Anzahl an Auflösungen in einen Entwicklungsalbtraum ausarten. Kaum eine Aussage in diesem Buch ist daher so absolut gemeint wie diese: Entwickeln Sie unter keinen Umständen für spezifische Auflösungen.

Flüssige Layouts zeichnen sich durch ihre Möglichkeit aus, bei Notwendigkeit größer oder kleiner zu werden und die enthaltenen Inhalte entsprechend neu auszurichten. Der Windows 8 Start Screen ist ein Beispiel dafür: Der komplette Bildschirm wird immer optimal ausgenutzt, um Kacheln anzuzeigen. Wird der Bildschirm größer, werden mehrere Reihen dargestellt; wird er kleiner, werden weniger Reihen angezeigt. Im klassischen Webdesign wird diese Vorgehensweise auch als *Responsive Web Design* bezeichnet.

Um Layouts flüssig zu gestalten, müssen grundsätzlich statt pixelbasierter Angaben proportionale Angaben verwendet werden. HTML5 und viele der neueren CSS-Werkzeuge unterstützen die Entwicklung von flüssigen Layouts – so etwa die in Kapitel 4 beschriebenen Kernwerkzeuge des Grid Layouts (`-ms-grid`), das Flexbox-Layout (`-ms-flexbox`) sowie Media Queries. Gleichzeitig sind viele der in `WinJS.UI` enthaltenen UI-Elemente bereits von Haus aus fit für flüssige Layouts: Das `WinJS.UI.ListView` etwa nutzt den ihm zur Verfügung gestellten Platz immer optimal aus. Ist ein `ListView` das einzige Control auf einer App-Seite, muss dessen Größe schlicht fix auf 100% Höhe und 100% Breite gesetzt werden. Um die geschickte Aufteilung des variabel zur Verfügung stehenden Platzes kümmert sich das System vollautomatisch.

Wo aus technischen Gründen kein flüssiges Layout eingesetzt werden kann – etwa weil man mit Inhalten zu tun hat, die über ein festes Seitenverhältnis verfügen –, hilft die `ViewBox`. Das in Kapitel 6 näher beschriebene WinJS-Control `WinJS.UI.ViewBox` ist ein Hostelement und skaliert

ein einzelnes untergeordnetes DOM-Element so, dass es den verfügbaren Platz bestmöglich ausfüllt, ohne das Seitenverhältnis zu beeinflussen. Auch hier ist das Vorgehen ähnlich wie bei dem `ListView` einfach: Soll beispielsweise ein Video angezeigt werden, ist es vollkommen ausreichend, der `ViewBox` 100% des Viewports zuzuweisen und das Video als untergeordnetes DOM-Element in der `ViewBox` zu verschachteln. Der optimalen Anzeige des Videos für den jeweils vom Benutzer verwendeten Bildschirm nimmt sich das System selbst an.

Grundsätzlich sind flüssige Layouts keine große Neuerung, sie sind aber aufgrund der hohen Qualitätsansprüche an Windows 8 Apps in der Entwicklung von etwas höherer Bedeutung und werden durch WinJS-Werkzeuge unterstützt. Eine echte Neuerung hingegen ist die Pixelskalierung.

Pixelskalierung: Ein perfektes Layout für jede DPI-Dichte

Die Problemstellung ist einfach: Gerade bei touchbasierten Geräten ist die physische Größe eines Buttons für die Nutzung entscheidend. Durch die zunehmende Bandbreite bei DPI-Werten auf dem Markt ist es jedoch unmöglich geworden, mit Gewissheit anzugeben, wie groß ein 200 Pixel breiter Button physisch sein wird.

Windows 8 skaliert Apps automatisch, um dieses Problem zu beheben. Dabei werden drei Stufen verwendet: 100%, 140% und 180%. Die unterste Stufe ohne Skalierung wird in der Entwicklung verwendet. Wird die App auf einem Bildschirm mit einer Auflösung von 1920 × 1080 Pixel oder höher und einem DPI-Wert von mindestens 174 geöffnet, skaliert Windows 8 die App automatisch auf 140%. Liegt die Auflösung bei 2560 × 1440 Pixel oder höher und der DPI-Wert mindestens bei 240, wird die App dagegen auf 180% skaliert.

Technisch ist die Unterstützung dieser Skalierung seitens der Entwicklung einfach: Sämtliche HTML-Elemente, die direkt von der Engine gezeichnet werden – also Text, Flächen, Formen oder Canvas-Bereiche –, werden automatisch bestmöglich und ohne Qualitätsverlust skaliert. Ebenso vollautomatisch erfolgt die Skalierung für das vektorbasierte Bildformat SVG, welches aus diesem Grund für die Entwicklung grundsätzlich empfohlen wird. Bitmap-Dateien, die nicht als SVG-Dateien hinterlegt werden können, können dem App-Paket in drei Größen mitgegeben werden. Dabei verwendet Windows 8 eine einfache Namenskonvention. Die Skalierung wird dabei wahlweise entweder im Dateinamen oder aber im Ordnernamen durch einen Zusatz angegeben. Windows 8 wählt dann beim Rendering automatisch die passende Datei aus:

```
meineBilder\meinBild.scale-100.jpg
meineBilder\meinBild.scale-140.jpg
meineBilder\meinBild.scale-180.jpg

meineBilder\scale-100\meinBild.jpg
meineBilder\scale-140\meinBild.jpg
meineBilder\scale-180\meinBild.jpg
```

Listing 8.1: Namenskonvention für Bilder in einem Nicht-Vektorformat

Im Code wird die Datei referenziert, als existiere die Skalierung nicht. Die im Beispiel verwendete Datei *meinBild.jpg* würde man für beide Namenskonventionen als `<img src="meineBilder/meinBild.jpg" width="100px" height="100px" />` referenzieren.

Bilder, die aus dem Web geladen werden sollen, können nicht mit der automatischen Namenskonvention arbeiten. Immerhin lässt sich per CSS Media Queries feststellen, mit welcher DPI-Auflösung der Viewport gerade arbeitet:

```
@media all and (max-resolution: 173dpi){
    /* Skalierung: 100% */
    #meinDiv {
        background-image: url('http://www.meinweb.de/meinBild-100.jpg');
    }
}
@media all and (min-resolution: 174dpi) {
    /* Skalierung: 140% */
    #meinDiv {
        background-image: url('http://www.meinweb.de/meinBild-140.jpg');
    }
}
@media all and (min-resolution: 240dpi) {
    /* Skalierung: 180% */
    #meinDiv {
        background-image: url('http://www.meinweb.de/meinBild-180.jpg');
    }
}
```

Listing 8.2: CSS Media Queries mit Zugriff auf DPI-Wert

Die Entwicklung für unterschiedlichste Bildschirmauflösungen und -größen wird durch den bei Visual Studio 2012 mitgelieferten Simulator unterstützt, welcher bei einem Klick auf das kleine Bildschirmsymbol in der rechten Symbolleiste verschiedene Auflösungen nebst der dazugehörigen Bildschirmgröße anbietet (Abbildung 8.1). Im Idealfall sollte sich die Ansicht nur bei einem Wechsel der Bildschirmgröße, nicht aber bei einem Wechsel der Auflösung verändern.

Zusammenfassend sind bei der Entwicklung der eigenen App nur wenige »Dos & Don'ts« zu beachten:

- Nutzen Sie SVG-Dateien, wo dies möglich ist
- Stellen Sie skalierte Größen von JPGs, PNGs und GIFs bereit
- Verwenden Sie CSS Media Queries, um für einzelne Bereiche in die Skalierung einzugreifen
- Setzen Sie den Simulator ein, um bereits am eigenen Bildschirm zu überprüfen, ob Ihre App die Skalierung ohne Beeinträchtigung oder gar Zerstörung des Layouts übersteht

Abbildung 8.1: Der Simulator und die Möglichkeit, verschiedene Auflösungen zu testen

Snap View & Fill View

8.2

Windows 8 erlaubt es Anwendern, eine App in einer Miniansicht am Bildschirmrand anzudocken, während der Großteil des Bildschirms von einer weiteren App eingenommen wird (Abbildung 8.2). Microsoft verlangt von Entwicklern die Unterstützung dieses Features.

Abbildung 8.2: Fill View mit der App »Karten« (links) und Snap View mit der App »MetroTwit«

Dabei ist der sogenannte *Fill View*, also die größere Variante der beiden Ansichten, im Regelfall keine gesonderte Überlegung wert: Das Snap View-Feature erfordert eine Mindestbildschirmbreite von 1366 Pixel, während die Anzeige einer einzelnen App mindestens 1024

Pixel voraussetzt. Apps, welche die kleinere Windows 8-Minimalauflösung von 1024 × 768 Pixel unterstützen, sind im Regelfall auch für den Fill View gewappnet.

Der Snap View ist bei einer Skalierung von 100 Prozent 320 Pixel breit – und im Gegensatz zum Fill View in seiner Breite statisch. Für eine erfolgreiche Zertifizierung für den Windows Store muss eine App den Snap View unter Berücksichtigung einiger User Experience-Richtlinien unterstützen:

1. Der State der App muss beim Wechsel zwischen den verschiedenen Ansichtsarten unangetastet bleiben. Selbst wenn Funktionalität vorübergehend außer Kraft gesetzt werden muss, sollten Nutzereingaben oder nicht gespeicherte Konfigurationen durch einen bloßen Wechsel der Ansicht nicht verändert werden oder gar verloren gehen.
2. Die Funktionalität im Snap View sollte weitestgehend mit der Funktionalität im Fill View identisch sein. Es wird empfohlen, dem geringer werdenden Platz im Snap View mit geschicktem Layout entgegenzuwirken, anstatt Features vorübergehend zu deaktivieren. Sollten einige Interaktionsmöglichkeiten im Snap View schlicht nicht angemessen abgebildet werden können, so kann man dem Nutzer bei dem Versuch, solch ein Feature zu nutzen, anbieten, die App aus dem Snap View zu »befreien«. Wichtig ist hier, den Nutzer einzubeziehen und nicht ungefragt einen »Unsnap« durchzuführen.
3. Wo wir gerade vom programmtechnischen »Unsnap« sprechen: Dieser sollte den notwendigen Situationen vorbehalten bleiben und keinesfalls genutzt werden, die App in den Vordergrund zu drängen. Ebenso sollte eine App keine User Interface-Elemente wie Buttons anbieten, um den Snap View zu verlassen – diese Aufgabe bleibt dem breiten schwarzen Balken zwischen den Apps vorbehalten.

Als Entwickler haben Sie zwei Möglichkeiten, auf den Snap View zu reagieren: Mit Media Queries können für die verschiedenen View States eigene CSS-Regeln gelten. Alternativ lässt sich mit einem Event Listener programmtechnisch reagieren und beispielsweise auf eine andere Seite navigieren.

Snap View, Fill View, Full View und Media Queries

Windows 8 implementiert für die Media Queries eine eigene Eigenschaft namens `-ms-view-state`, welche über folgende Werte verfügt:

- `fullscreen-landscape` Die App ist im Vollbildschirmmodus, der Bildschirm wird horizontal gehalten (Querformat)
- `fullscreen-portrait` Die App ist im Vollbildschirmmodus, der Bildschirm wird vertikal gehalten (Hochformat, wie ein Taschenbuch)

- `filled` Eine weitere App ist im Snap View, die eigene App nimmt den Großteil der Bildschirmfläche ein
- `snapped` Die eigene App ist im Snap View, eine andere App nimmt den Großteil der Bildschirmfläche ein

Damit lassen sich eigene CSS-Regeln definieren, die in vielen Fällen ausreichen, um Apps fit für den Snap View zu machen:

```
@media screen and (-ms-view-state: fullscreen-landscape) {
  .meineRegel {
    width: 100%;
  }
}
@media screen and (-ms-view-state: filled) {
  .meineRegel {
    width: calc(100% - 320px);
  }
}
@media screen and (-ms-view-state: snapped) {
  .meineRegel {
    width: 320px;
  }
}
@media screen and (-ms-view-state: fullscreen-portrait) {
  .meineRegel {
    height: 100%;
  }
}
```

Listing 8.3: Media Queries und View States

View States und Event Listener

In Windows 8 verfügt das Windows-Objekt ebenso wie im »normalen« Browser über einen `resize`-Event, welcher ausgelöst wird, sobald die App ihre Größe verändert – etwa, weil sie selbst oder eine andere App in den Snap View geschoben wurde. In der Windows-API `Windows.UI.ViewManagement.ApplicationView.value` findet sich der aktuelle ViewState als Objekt wieder. Die potenziellen Rückgabewerte sind Teile einer Enumeration, welche in `Windows.UI.ViewManagement.ApplicationViewState` enthalten sind. Der nachfolgende Code zeigt, wie man mit einer einfachen Abfrage auf die unterschiedlichen ViewState-Wechsel reagieren kann:

```
function meinViewState() {
  var aktuellerViewState = Windows.UI.ViewManagement.ApplicationView.value;
  var meineViewStates = Windows.UI.ViewManagement.ApplicationViewState;
```

```
  switch (aktuellerViewState) {
    case meineViewStates.snapped:
      // Auf "Snap View" reagieren
      break;
    case meineViewStates.filled:
      // Auf "Fill View" reagieren
      break;
    case meineViewStates.fullScreenLandscape:
      // Auf "Full View Landscape" reagieren
      break;
    case meineViewStates.fullScreenPortrait:
      // Auf "Full View Portrait" reagieren
      break;
  }
}

window.addEventListener("load", meinViewState);
window.addEventListener("resize", meinViewState);
```

Listing 8.4: Reaktion auf View State-Wechsel mit einem Event Listener und der Abfrage des ViewState-Objekts

Der Namespace `ApplicationView` hält darüber hinaus die Methode bereit, mit der eine zuvor in den Snap View verbannte App programmtechnisch versuchen kann, sich wieder in den Fill View zu schalten. Die Methode `Windows.UI.ViewManagement.ApplicationView.tryUnsnap()` sollte jedoch nur entsprechend den oben erwähnten Richtlinien verwendet werden, also nur dann, wenn es im Sinne fehlender Funktionalität notwendig ist, den Snap View zu verlassen.

8.3 Geräteorientierung

Moderne Tablets können gedreht werden – ein Umstand, mit dem sich klassische Windows-Anwendungsentwickler bislang nicht befassen mussten. Unter Windows 8 bleibt es zwar dem Entwickler überlassen, ob er auf einen Ausrichtungswechsel reagieren möchte, sicherlich gehört es jedoch zum guten Ton einer hochwertigen App, dass sich diese den Vorlieben des Nutzers anpasst. Ähnlich wie beim Snap View lässt sich auf eine Änderung der Geräteorientierung sowohl mit Media Queries als auch mit einem Event Listener reagieren. Wird bei der Entwicklung nicht explizit eingeschritten, wird Windows 8 von selbst versuchen, die aktuelle Ansicht bei einer Gerätedrehung den neuen Bedingungen anzupassen. Gerade wenn das Layout komplett flüssig ist, sollte dies im Regelfall auch glücken.

Die Media Query-Eigenschaft `-ms-view-state` verfügt über die beiden potenziellen Werte `fullscreen-landscape` und `fullscreen-portrait`. Da im Portrait-Modus kein Snap View unterstützt wird, müssen Entwickler als zusätzliche Ansicht lediglich den Modus `fullscreen-portrait` unterstützen. Die Nutzung der Media Queries erlaubt es, kleine Ungereimtheiten, die aufgrund des geänderten Seitenverhältnisses auftreten, mit einfachen CSS-Korrekturen zu beheben:

```
@media screen and (-ms-view-state: fullscreen-portrait) {
  .meineRegel {
    height: 100%;
  }
}
```

Listing 8.5: CSS Media Queries und eine Regel, die nur in der Portrait-Geräteausrichtung greift

Soll programmtechnisch auf eine Änderung reagiert werden, kann dies ebenfalls wie beim Snap View über den `resize`-Event des `Windows`-Objekts sowie den im Namespace `Windows.UI.ViewManagement.ApplicationView` vorhandenen Angaben geschehen:

```
function meinViewState() {
  var aktuellerViewState = Windows.UI.ViewManagement.ApplicationView.value;
  var meineViewStates = Windows.UI.ViewManagement.ApplicationViewState;

  switch (aktuellerViewState) {
    case meineViewStates.fullScreenLandscape:
      // Auf "Full View Landscape" reagieren
      break;
    case meineViewStates.fullScreenPortrait:
      // Auf "Full View Portrait" reagieren
      break;
  }
}

window.addEventListener("load", meinViewState);
window.addEventListener("resize", meinViewState);
```

Listing 8.6: Reaktion auf Wechsel zwischen Portrait- und Landscape-Modus

Die Animation des Orientierungswechsels übernimmt Windows automatisch. Ein Eingreifen seitens des Entwicklers ist hier nicht möglich. Wer etwas mehr Mühe investieren möchte, kann an dieser Stelle auch an die User Experience denken und berücksichtigen, dass Nutzer das Scrollen an der längeren Achse grundsätzlich bevorzugen. Das bedeutet zunächst, dass im Landscape-Modus wie gewohnt horizontal gescrollt wird. Zur Optimierung des User Experience kann passend dazu im Portrait-Modus dann vertikal gescrollt werden. Dieses Feature ist jedoch seitens Microsoft keine Pflicht – ebenso wenig wie eine Implementierung des Semantic Zooms, der die Übersicht in einer App massiv verbessern kann und um den es im Folgenden geht.

8.4 Semantic Zoom

Der Semantic Zoom erlaubt es Anwendern, bei Toucheingabe über die Zweifinger-Pinch-Geste und bei Mausverwendung über ein kleines Minussymbol am unteren rechten Bildschirmrand die aktuelle Ansicht drastisch zu verkleinern, um auf die Schnelle mehr Übersicht in der aktuellen Ansicht zu erhalten. Ein populäres Beispiel ist der Windows 8 Start Screen

(Abbildung 8.3), der auch für den Fall, dass unzählige Apps vorhanden sind, einen schnellen Sprung in die hintersten Bereiche des Menüs ermöglicht.

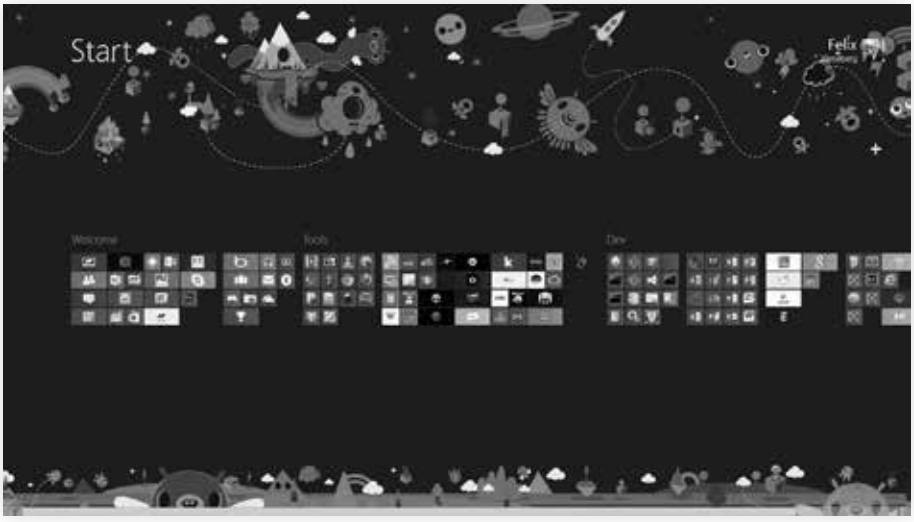

Abbildung 8.3: Semantic Zoom im Startbildschirm

Bei der Umsetzung des Semantic Zooms gibt es einige Regeln zu beachten:

1. Ein Semantic Zoom betrifft grundsätzlich die gesamte Bildschirmfläche der App, darf also nicht verschachtelt werden. Ebenso ist die anfängliche Ansicht immer die »herangezoomte«, also größere Ansicht. Der Wechsel in den Semantic Zoom führt zu mehr Übersicht.

2. Der aktuelle Scope, also der Geltungsbereich der aktuellen Ansicht, sollte sich nicht ändern. Wenn der Anwender gerade eine lange Liste mit aktuellen Filmen sieht, sollte er auch nach Anwendung des Semantic Zooms weiterhin noch eine Filmliste vor sich haben – auch wenn jetzt die Filme nur noch stilisiert angezeigt werden und stattdessen deren Kategorien im Vordergrund stehen. Die zur Verfügung stehende Funktionalität darf sich aber durchaus ändern – so wie App-Gruppen auf dem Windows 8 Start Screen nur im Semantic Zoom benannt werden können.

3. Ist der Semantic Zoom aktiviert, soll der Nutzer mehr Übersicht über den Inhalt erhalten. Das bedeutet, dass der gesamte Inhalt auf maximal drei Bildschirmflächen passen sollte.

4. Technisch wird der Semantic Zoom in HTML5/JavaScript-Apps mit dem eigenen Control `WinJS.UI.SemanticZoom` umgesetzt. Das Control, dessen Methoden und Eigenschaften bereits in Kapitel 6 beschrieben wurden, basiert auf einem simplen Prinzip: Das Control enthält zwei weitere Controls, wobei das erste die herangezoomte und das zweite die herausgezoomte Ansicht darstellt.

```
<div data-win-control="WinJS.UI.SemanticZoom">
    <!-- Herangezoomte Ansicht -->
    <div id="herangezoomt" data-win-control="WinJS.UI.ListView"></div>
```

```
    <!-- The control that provides the zoomed-out view goes here. -->
    <div id="herausgezoomt" data-win-control="WinJS.UI.ListView"></div>
</div>
```

Listing 8.7: Nutzung des WinJS.UI.SemanticZoom-*Controls*

Wichtig ist, dass die enthaltenen Controls das `IZoomableView`-Interface unterstützen. Wer sich viel Arbeit machen möchte, kann das Interface zwar selbst implementieren, es gibt jedoch kaum Fälle, in denen das einzige WinJS-Control mit `IZoomableView`-Interface – das `ListView` – die gewünschte Funktionalität nicht abdecken kann. Das Interface ist notwendig, um die Verbindung zwischen den beiden angezeigten Controls herzustellen. Immerhin handelt es sich im Wesentlichen um zwei unterschiedliche Ansichten derselben Daten: Wer in der Semantic Zoom-Übersicht auf einen Gruppentitel tippt, möchte auch in der herangezoomten Ansicht direkt zu der entsprechenden Gruppe springen.

SemanticZoom-Implementation mit ListView-Gruppen

Ein `SemanticZoom`–Control wird mit Windows 8-Bordmitteln über zwei `ListView`'-Controls realisiert, die auf dasselbe Set an gruppierten Daten zugreifen. Dabei zeigt die herangezoomte Ansicht sämtliche Elemente an, während die herausgezoomte Semantic Zoom-Ansicht lediglich die Gruppen darstellt. Wird im Semantic Zoom-Modus eine Gruppe angetippt, sorgt das `IZoomableView`-Interface dafür, dass der Semantic Zoom verlassen und direkt zu der entsprechenden Gruppe gesprungen wird.

Gruppierte Daten lassen sich über die Methode `createGrouped()` eines `WinJS.Binding.List`-Objektes erstellen. Die Methode akzeptiert drei Parameter:

1. `groupKey` Eine Gruppenschlüsselfunktion, die für jedes Element in der Liste mit dem Element als Parameter aufgerufen wird. Die Methode sollte einen String zurückgeben, der die Gruppe für das jeweilige Element repräsentiert. Es ist wichtig, dass die Funktion bei mehreren Aufrufen für dasselbe Element auch denselben Wert zurückgibt – wenn sich Elemente durch externen Einfluss in dieser Hinsicht ändern, muss die `WinJS.Binding.List`-Methode `notifyMutated(index)` aufgerufen werden – mit dem Index des geänderten Elements als Parameter.

2. `groupData` Eine Gruppendatenfunktion, die für jede Gruppe einmalig aufgerufen wird und ein beliebiges Element aus der Liste als Parameter enthält. Als Rückgabewert werden die Daten erwartet, welche die `WinJS.Binding.List` als Daten aufweist, wenn das Suffix `.groups` angegeben wird. Im Klartext: Eine `WinJS.Binding.List` mit dem Namen `meineListe` enthält unter `meineListe.dataSource` die Datenquelle für alle Elemente, während `meineListe.groups.dataSource` nur jene Daten aufweist, die von der Gruppendatenfunktion zurückgegeben wurden.

3. `groupSorter` Diese optionale Gruppensortierfunktion ist nur von Bedeutung, wenn die Gruppen programmtechnisch sortiert werden müssen. Sie wird mit jeweils zwei Gruppenschlüsseln als Parameter aufgerufen und muss einen numerischen Wert zurückgeben. Ein negativer Wert bedeutet, dass der zweite Gruppenschlüssel nach dem ersten

Gruppenschlüssel einsortiert werden sollte, ein positiver Wert dagegen signalisiert, dass der erste Gruppenschlüssel nach dem zweiten einsortiert werden sollte. Eine 0 bedeutet, dass die beiden Gruppen gleichwertig sind.

4. Das nachfolgende Codebeispiel zeigt eine Datenliste mit Namen, die in drei Gruppen (Team A, Team B und Team C) eingeteilt sind – und wie die `createGrouped()`-Methode genutzt wird. Zunächst der HTML-Code:

```
<div id="meinSemanticZoom" data-win-control="WinJS.UI.SemanticZoom">
    <div id="herangezoomt" data-win-control="WinJS.UI.ListView"></div>
    <div id="herausgezoomt" data-win-control="WinJS.UI.ListView"></div>
</div>

<div id="gruppenTemplate" data-win-control="WinJS.Binding.Template">
    <h2 data-win-bind="innerText: gruppe"></h2>
</div>

<div id="elementeTemplate" data-win-control="WinJS.Binding.Template">
    <h2 data-win-bind="innerText: name"></h2>
</div>

<div id="gruppenHeaderTemplate" data-win-control="WinJS.Binding.Template">
    <h1 data-win-bind="innerText: gruppe"></h1>
</div>
```

Listing 8.8: HTML-Code für ein SemanticZoom *mit gruppierten* ListView-*Controls*

Wie zu sehen ist, werden zwei `ListView`-Controls innerhalb des `SemanticZoom`-Controls definiert, ebenso wie drei Templates – eines für die Elemente, eines für die Gruppentitel in der Elementeansicht und eines für die Gruppen im `SemanticZoom`. Nun der JavaScript-Code, der für die Umsetzung notwendig ist und auf die erwähnte `createGrouped()`-Methode zurückgreift:

```
// Die beiden ListView-Controls
var herangezoomt = document.getElementById("herangezoomt");
var herausgezoomt = document.getElementById("herausgezoomt");

var meineDaten = [
    { name: "Marlena Zuba", gruppe: "Team A"},
    { name: "Johannes Sieben", gruppe: "Team A"},
    { name: "Nadja Nonnen", gruppe: "Team A"},
    { name: "Katha Sieg", gruppe: "Team A"},
    { name: "Anika Kunz", gruppe: "Team A"},
    { name: "Sarah Bellenbaum", gruppe: "Team B"},
    { name: "Christina Lorke", gruppe: "Team B"},
    { name: "Marcel Malmedy", gruppe: "Team B"},
    { name: "Tim Baumers", gruppe: "Team B"},
    { name: "Jens Paschke", gruppe: "Team B"},
```

```
    { name: "Tom Wendel", gruppe: "Team C"},
    { name: "Patric Boscolo", gruppe: "Team C"},
    { name: "Olivia Klose", gruppe: "Team C"},
    { name: "Heike Ritter", gruppe: "Team C"}
];
var meineBindingList = new WinJS.Binding.List(meineDaten);
var meineGruppenBindingList = meineBindingList.createGrouped(
        function gruppenKey(i) { return i.gruppe; },
        function gruppenDaten(i) { return { gruppe: i.gruppe }; }
    );

herangezoomt.winControl.itemDataSource = meineGruppenBindingList.dataSource;
herangezoomt.winControl.groupDataSource = meineGruppenBindingList.groups.dataSource;
herangezoomt.winControl.itemTemplate = elementeTemplate;
herangezoomt.winControl.groupHeaderTemplate = gruppenHeaderTemplate;

herausgezoomt.winControl.itemDataSource = meineGruppenBindingList.groups.dataSource;
herausgezoomt.winControl.itemTemplate = gruppenTemplate;
```

Listing 8.9: JavaScript für ein SemanticZoom *mit gruppierten* ListView*-Controls*

Das komplette »Drumherum«, also etwa die verschiedenen Aktivierungsmethoden des Semantic Zooms je nach Eingabemethode, die Animationen oder die Verbindung der beiden Datensätze wird vollautomatisch von Windows 8 verwaltet.

Der Semantic Zoom schließt damit den theoretischen Überblick über die verschiedenen Ansichtsoptimierungen ab. Im Folgenden steht wieder die *Eierlegende Wollmilchsau* im Vordergrund, in der nun auch sichergestellt werden soll, dass die App in jeder Situation eine gute Figur macht.

8.5 App zum Mitentwickeln: Optimierung für verschiedene Ansichten

Die im letzten Kapitel erstellte Version der *Eierlegenden Wollmilchsau* gibt in manchen Situationen ein trauriges Bild ab: Ein Semantic Zoom ist nicht implementiert und der Snap View ist ein Albtraum. In zwei Schritten sollen diese Umstände behoben werden: Zunächst wird ein `SemanticZoom`-Control sowie Gruppendaten implementiert, um anschließend die App fit für verschiedene Ansichten zu machen.

Schritt 1: SemanticZoom implementieren

1. Zunächst müssen die Daten in verschiedene Gruppen aufgeteilt werden. Da mein bescheidener Blog nicht über ausreichend Inhalte verfügt, um im angemessener Anzahl Kategorien bereitzuhalten, habe ich mich dafür entschieden, die Artikel nach dem Monat ihres Erscheinens zu sortieren. Dafür muss die Klasse `WollmilchRSS` in der Datei

rssData.js erweitert werden. Zunächst wird dafür gesorgt, dass bei der Verarbeitung der heruntergeladenen Daten das Erscheinungsjahr sowie der Monat des Artikels mitgespeichert werden. Innerhalb der Funktion feedAbholen() wird jeder Artikel einzeln als Array zusammengesetzt, um anschließend in die _bindingList gepusht zu werden. Dort werden einige Zeilen hinzugefügt, die den Artikel um die Eigenschaften gruppe, monat und jahr erweitern:

```
for (var n = 0; n < artikel.length; n++) {
    var einzelArtikel = {};

    einzelArtikel.titel = artikel[n].querySelector("title").textContent;
    einzelArtikel.link = artikel[n].querySelector("link").textContent;
    einzelArtikel.inhalt = artikel[n].querySelector("description").textContent;
    einzelArtikel.vorschautext = einzelArtikel.inhalt.substring(0, 100) + "...";

    // Neu
    var monatsNamen = ["Januar", "Februar", "März", "April", "Mai", "Juni", "Juli",
        "August", "September", "Oktober", "November", "Dezember"];
    var artikelDatum = new Date(artikel[n].querySelector("pubDate").textContent);
    einzelArtikel.gruppe = monatsNamen[artikelDatum.getMonth()] + " " +
        artikelDatum.getFullYear();
    einzelArtikel.monat = artikelDatum.getMonth();
    einzelArtikel.jahr = artikelDatum.getFullYear();
    // Ende Neu

    var thumbs = artikel[n].querySelectorAll("enclosure");

    if (thumbs.length > 0){
        einzelArtikel.thumbnail =
            "url('" + thumbs[0].attributes.getNamedItem("url").textContent + "')";
        that.bindingList.push(einzelArtikel);
    }
}
```

Listing 8.10: Erweiterung der Einzelartikelverarbeitung in der Funktion feedAbholen() *der Klasse* WollmilchRSS

2 Gleichzeitig muss die Klasse um eine gruppierte Liste erweitert werden. Im Konstruktor function WollmilchRSS(url) wird deshalb die Definition um eine solche gruppierte Liste erweitert. Zunächst wird die Definition this.dataSource auskommentiert, da sie momentan noch auf die nicht gruppierte Liste zeigt. Anschließend wird eine _gruppierteBindingList erstellt, welche die createGrouped()-Methode der »gewöhnlichen« Binding.List nutzt. Dort werden auch drei Funktionen implementiert, die aus den im vorigen Schritt zusätzlich gewonnenen Daten saubere Gruppen erzeugen – und auch sicherstellen, dass die Reihenfolge der Gruppen korrekt ist.

```
WollmilchRSS: WinJS.Class.define(
    function WollmilchRSS(url) {

        var _dataArray = [];
        var _bindingList = new WinJS.Binding.List();

        this.url = url;
        this.bindingList = _bindingList;
        //this.dataSource = _bindingList.dataSource;

        //Neu
        var _gruppierteBindingList = _bindingList.createGrouped(
            function gruppenKey(i) { return i.monat + "_" + i.jahr; },
            function gruppenDaten(i) { return { gruppe: i.gruppe, monat: i.monat,
                jahr: i.jahr }; },
            function gruppenVergleich(a, b) {
                var a = a.split("_");
                var b = b.split("_");
                var aJahr = a[1];
                var aMonat = a[0];
                var bJahr = b[1];
                var bMonat = b[0];

                if (aJahr > bJahr) {
                    return -1;
                }
                else if (aJahr == bJahr) {
                    if (aMonat > bMonat) { return 1 }
                    else if (aMonat == bMonat) { return 0 }
                    else { return -1 }
                }
                else if (aJahr < bJahr) {
                    return 1;
                }
            }
            );

        this.dataSource = _gruppierteBindingList.dataSource;
        this.gruppierteBindingList = _gruppierteBindingList;
        this.gruppenBindingList = _gruppierteBindingList.groups;
        this.gruppenDataSource = _gruppierteBindingList.groups.dataSource;
        //Ende Neu
    },
```

Listing 8.11: *Erweiterung des Konstruktors der Klasse* WollmilchRSS

3. Gleichzeitig muss sichergestellt werden, dass im Zuge des Hinzufügens der neuen Listen kein Datendurcheinander entsteht, sobald die Funktion mehrfach aufgerufen wird. Aus diesem Grund werden die Listen ab sofort geleert, sobald der Anwender die Methode `feedAbholen()` aufruft. Fügen Sie dafür im Erfolgshandler von `WinJS.xhr()` im `else`-Abschnitt `that.bindingList.length = 0;` und `that.gruppierteBindingList.length = 0;` ein, um beide Listen vor einer neuer Datenbefüllung zu leeren.

Listing 8.12: Leeren beider Listen im Erfolgshandler der Funktion WinJS.xhr() *in* feedAbholen()

```
} else {

    that.bindingList.length = 0;
    that.gruppierteBindingList.length = 0;

    var artikel = daten.responseXML.querySelectorAll("item");

for (var n = 0; n < artikel.length; n++) {
```

4. Nun geht es daran, den HTML-Code zu erweitern. Bisher wurde ein einfaches `ListView` verwendet, welches nun zusammen mit einem weiteren `ListView` für den Semantic Zoom in das Control `WinJS.UI.SemanticZoom` gepackt werden muss. Öffnen Sie die Datei *home.html* und verändern Sie den Code der `<section>` wie folgt:

Listing 8.13: Integration des bisher verwendeten ListView-*Controls innerhalb eines* SemanticZoom-*Controls*

```
<section aria-label="Main content" role="main">
    <!-- Neu -->
    <div id="meinSemanticZoom" data-win-control="WinJS.UI.SemanticZoom">
        <div id="meinListView" data-win-control="WinJS.UI.ListView"
            data-win-options="{ itemTemplate: select('#meineElementVorlage'),
            groupHeaderTemplate: select('#meineGruppenHeaderVorlage'),
            selectionMode:'single' }"></div>
        <div id="meinSemanticListView" data-win-control="WinJS.UI.ListView"
            data-win-options="{ itemTemplate: select('#meineGruppenVorlage') }">
        </div>
    </div>
</section>
```

5. Außerdem müssen zwei weitere Vorlagen implementiert werden: erstens eine Elementvorlage für das `SemanticZoom-ListView`, in welchem lediglich die Monate zu sehen sein sollen. Zweitens wird noch eine Vorlage für die Gruppentitel benötigt. Im oberen Bereich der Datei *home.html* finden Sie die Vorlagen, diesen Bereich erweitern Sie wie folgt:

```
<!-- Neue Elementvorlagen -->
<div id="meineGruppenHeaderVorlage" data-win-control="WinJS.Binding.Template">
    <h2 data-win-bind="innerText: gruppe"></h2>
</div>
<div id="meineGruppenVorlage" data-win-control="WinJS.Binding.Template">
    <div class="einzelGruppenElement">
```

```
            <h2 class="einzelGruppenTitel" data-win-bind="innerText: gruppe"></h2>
        </div>
    </div>
```

Listing 8.14: Erweiterung der Elementvorlagen in der Datei home.html

6 Nun müssen den neuen `ListView`-*Controls* weitere Datenquellen zugewiesen werden. Das bislang schon verwendete `ListView` benötigt noch eine Datenquelle für die Gruppen, während das neue ListView für den SemanticZoom generell mit einer Datenquelle versorgt werden muss. Öffnen Sie die Datei *home.js* und erweitern Sie den Code direkt unterhalb der Definition der Variable `meinListView` wie folgt:

```
var meinListView = document.getElementById("meinListView").winControl;

//Neu
meinListView.groupDataSource = wollmilchRSS.gruppenDataSource;
meinListView.itemDataSource = wollmilchRSS.dataSource;

var meinSemanticListView = document.getElementById("meinSemanticListView").winControl;
meinSemanticListView.itemDataSource = wollmilchRSS.gruppenDataSource;

meinListView.onselectionchanged = selectionChangedHandler;
```

Listing 8.15: Zuweisung der neuen Datenquellen in home.js

Im für den `SemanticZoom` letzten Schritt müssen noch einige CSS-Anpassungen vorgenommen werden, damit die Optik auch stimmt. Öffnen Sie die Datei *home.css*: Hier muss für die Regel `#meinListView` eine kleine Änderung vorgenommen werden, außerdem werden zwei neue Regeln namens `#meinSemanticZoom` sowie `.einzelGruppenElement` benötigt. Außerdem soll die Gelegenheit genutzt werden, um der Section ihre 120px `margin` an der linken Seite zu rauben, damit Elemente vollständig nach links scrollen und nicht im »Nirgendwo« verschwinden. Da Gruppenheader automatisch für 70 Pixel `margin` an der linken Seite sorgen, müssen hier unterschiedliche »Scroll-Anfangspunkte« für die beiden `ListView`-Controls definiert werden.

```
.homepage section[role=main] {
    margin-left: 0px;
}

@media screen and (-ms-view-state: fullscreen-landscape) {
    #meinListView {
        height: 100%;
    }
    #meinListView .win-surface {
        margin-left: 50px !important;
    }
```

```
    #meinSemanticListView .win-surface {
        margin-left: 120px !important;
    }
    #meinSemanticZoom {
        height: 100%;
        width: 100%;
    }
}
.einzelGruppenElement {
    width: 250px;
    height: 100px;
    padding: 20px;
}
```

Listing 8.16: Erweiterung des CSS in home.css

7. Entfernen Sie in der Datei home.css außerdem – falls noch vorhanden – den folgenden Eintrag, er wird nur nicht mehr gebraucht, sondern führt jetzt auch zu einer unerwünschten Darstellung:

```
.win-item {
    height: 200px;
}
```

8. Herzlichen Glückwunsch, Sie haben erfolgreich einen Semantic Zoom implementiert! Wenn Sie die App starten, sollten Sie nicht nur ein gruppiertes `ListView`, sondern auch die `SemanticZoom`-Funktionalität vorfinden (Abbildungen 8.4 und 8.5).

Abbildung 8.4: Aktivierter Semantic Zoom…

Abbildung 8.5: …und gruppierte Elemente im ListView

Schritt 2: Optimierung für verschiedene Ansichten

1. Ein Test im Visual Studio-Simulator (Abbildung 8.5) offenbart, dass eine gesonderte Anpassung für verschiedene Auflösungen nicht notwendig ist. Da bislang nur in Windows 8 enthaltene Controls verwendet und keine groben Verletzungen der Richtlinien begangen wurden, kann sich die *Eierlegende Wollmilchsau* in puncto unterschiedliche Bildschirme und Auflösungen voll auf Windows 8 verlassen: Die App skaliert bei unterschiedlichen Auflösungen und Bildschirmgrößen hervorragend. Hier ist schon mal keine weitere Arbeit notwendig.

Abbildung 8.6: Vollautomatische Skalierung: Die App skaliert passend zur Auflösung

2. Weniger rosig ist die Situation in puncto Snap View: Nicht nur der Startbildschirm mit dem `ListView` sieht bescheiden aus, auch die Artikelansicht macht eine schlechte Figur (Abbildung 8.7).

Abbildung 8.7: Der Snap View-Modus gibt eine rundum schlechte Figur ab

3. Zunächst soll der Startbildschirm einer gesonderten Behandlung unterzogen werden. Hier werden beide `ListView`-Controls des Semantic Zooms anstelle eines Grid- ein Listenlayout erhalten, bei denen die einzelnen Elemente und Gruppen einfach untereinander angeordnet sind. Ebenso sollen für einzelne Elemente andere Templates verwendet werden, um dem geringeren Platz gerecht zu werden. Dafür müssen zunächst in der Datei *home.html* zwei neue Elementvorlagen integriert werden:

```
<!-- SnapView Elementvorlagen -->
<div id="meineSVElementVorlage" data-win-control="WinJS.Binding.Template">
    <div class="einzelSVElement">
        <h3 class="elementSVTitel" data-win-bind="innerText: titel"></h3>
    </div>
</div>
<div id="meineSVGruppenVorlage" data-win-control="WinJS.Binding.Template">
    <div class="einzelSVElement">
        <h2 class="elementSVTitel" data-win-bind="innerText: gruppe"></h2>
    </div>
</div>
```

Listing 8.17: Zusätzliche Elementvorlagen für die ListView*-Controls in* home.html

4. Anschließend müssen zusätzliche CSS-Anweisungen implementiert werden, die im Snap View für ein passendes Layout sorgen. Dafür können Media Queries genutzt werden – mittels der Abfrage `-ms-view-state: snapped` lassen sich Regeln definieren, die nur im Snap View greifen. Die neuen Regeln kümmern sich im Wesentlichen darum, dass Abstände im Snap View kleiner sind, um mehr Inhalte auf die geringe Fläche zu bekommen. Fügen Sie in der Datei *home.css* folgenden Abschnitt hinzu:

```
@media screen and (-ms-view-state: snapped) {
    .homepage section[role=main] {
        margin-left: 20px;
    }
    #meinListView .win-surface {
        margin-left: 5px !important;
    }
    #meinSemanticListView .win-surface {
        margin-left: 5px !important;
    }
    #meinSemanticZoom {
        height: 100%;
        width: 300px;
    }
    .einzelSVElement {
        width: 260px;
        height: 60px;
        padding: 5px;
    }
    .win-container {
        width: 270px !important;
        margin-left: 0px !important;
    }
    .titlearea {
        margin-top: 10px !important;
    }
    .fragment {
        -ms-grid-rows: 60px 1fr !important;
    }
}
```

Listing 8.18: Neue CSS-Regeln für die Datei home.css

5 Anschließend kann mit einem Event Listener in der Datei *home.js* auf einen Wechsel der Ansicht reagiert und dort das Layout der zwei `ListView`-Controls angepasst werden. Fügen Sie folgenden Code am Ende der `ready`-Funktion in *home.js* ein. Es handelt sich um eine Funktion, die den aktuellen ViewState abfragt und entsprechend die Elementvorlagen sowie die Layouteinstellung der beiden `ListView`-Controls anpasst:

```
// Neu: Resize handling
function meinViewState() {
    var aktuellerViewState = Windows.UI.ViewManagement.ApplicationView.value;
    var meineViewStates = Windows.UI.ViewManagement.ApplicationViewState;
```

```
        switch (aktuellerViewState) {
            case meineViewStates.snapped:
                // Auf "Snap View" reagieren
                meinListView.layout = new WinJS.UI.ListLayout;
                meinListView.itemTemplate =
                    document.getElementById("meineSVElementVorlage");
                meinSemanticListView.layout = new WinJS.UI.ListLayout;
                meinSemanticListView.itemTemplate =
                    document.getElementById("meineSVGruppenVorlage");
                break;
            default:
                // Auf alle anderen View States reagieren
                meinListView.layout = new WinJS.UI.GridLayout;
                meinListView.itemTemplate =
                    document.getElementById("meineElementVorlage");
                meinSemanticListView.layout = new WinJS.UI.GridLayout;
                meinSemanticListView.itemTemplate =
                    document.getElementById("meineGruppenVorlage");
                break;
        }
    }

    window.addEventListener("load", meinViewState);
    window.addEventListener("resize", meinViewState);
    meinViewState();
```

Listing 8.19: Event Listener, der auf eine Änderung im ViewState reagiert und die ListView*-Controls entsprechend anpasst*

6 Anschließend kann die Detailseite angegangen werden. Hier gibt es nur wenig zu tun: Einige wenige CSS-Anpassungen sollten die Abstände korrigieren, während der Titel im Snap View aufgrund des engen Platzes ein wenig verloren wirkt und in dieser besonderen Situation eigentlich unterhalb des Zurück-Buttons angebracht werden sollte. Zunächst die CSS-Anpassung, hierbei fügen Sie folgende Regeln in die Datei *detailview.css* ein:

```
@media screen and (-ms-view-state: snapped) {
    .detailview section[role=main] {
        margin-left: 20px;
        margin-right: 20px;
        width: calc(100% - 40px);
    }
    .fragment header[role=banner] .win-backbutton {
        margin-top: 28px;
    }
    .fragment {
        -ms-grid-rows: 60px 1fr !important;
    }
```

```
    .titlearea .pagetitle {
        white-space: normal;
        overflow: visible !important;
    }
    #titelSV {
        font-size: large;
    }
}
```

Listing 8.20: Zusätzliche CSS-Regeln für die Datei detailview.css

7 Das Titelproblem lässt sich am einfachsten mit einem zweiten Titel lösen, der im Content-Bereich platziert wird. Anstatt also den einen Titel je nach Ansichtsmodus mit hunderten CSS-Regeln zurechtzubiegen, soll jetzt einfach passend zum Ansichtsmodus der entsprechende Titel sichtbar geschaltet werden. Fügen Sie zunächst den zusätzlichen Titel in der Datei *detailview.html* ein:

```
<div class="detailview fragment">
    <header aria-label="Header content" role="banner">
       […]
    </header>
    <section aria-label="Main content" role="main">
        <span class="pagetitle" id="titelSV"></span>
        <div id="inhalt"></div>
    </section>
</div>
```

Listing 8.21: Zusätzlicher Titel in detailview.html

8 Jetzt wird ähnlich wie in *home.js* ein Event Listener genutzt, um je nach Ansichtsart zwischen den beiden Titeln hin- und herzuschalten. Passen Sie die `ready`-Funktion in der Datei *detailview.js* wie folgt an:

```
ready: function (element, options) {

    titel.innerText = rssData.aktuellerArtikel.titel;
    titelSV.innerText = rssData.aktuellerArtikel.titel;
    inhalt.innerText = rssData.aktuellerArtikel.inhalt;

    // Neu: Resize handling

    function meinViewState() {
        var aktuellerViewState = Windows.UI.ViewManagement.ApplicationView.value;
        var meineViewStates = Windows.UI.ViewManagement.ApplicationViewState;

        switch (aktuellerViewState) {
            case meineViewStates.snapped:
                // Auf "Snap View" reagieren
                titelSV.style.visibility = "visible";
```

```
                titel.style.visibility = "collapse";
                break;
            default:
                // Auf alle anderen View States reagieren
                titelSV.style.visibility = "collapse";
                titel.style.visibility = "visible";
                break;
        }
    }

    window.addEventListener("load", meinViewState);
    window.addEventListener("resize", meinViewState);
    meinViewState();

},
```

Listing 8.22: Erweiterung der ready*-Funktion in* detailview.js *um einen Event Listener für ViewState-Änderungen*

9. Anschließend verfügen beide Ansichten über einen brauchbaren Snap View (Abbildung 8.8).

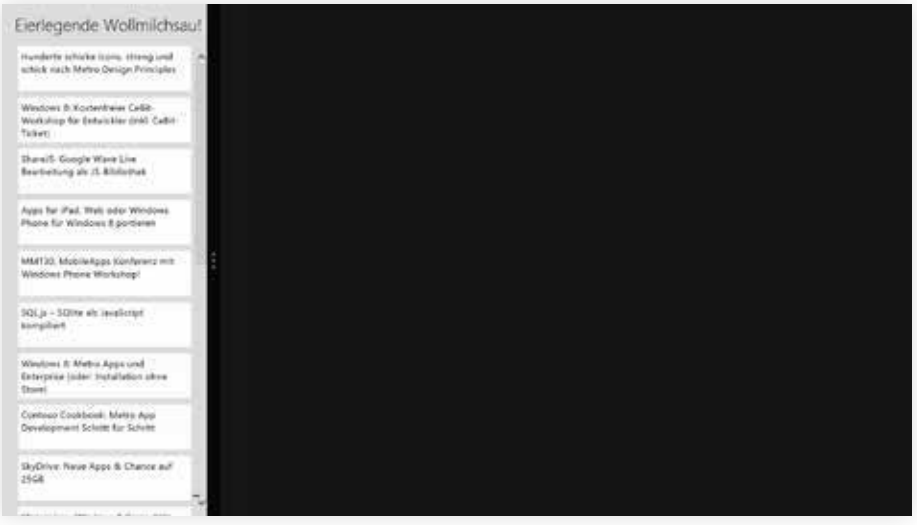

Abbildung 8.8: Snap View der Artikelübersicht

10. Im letzten Schritt soll auch noch die Portrait-Ansicht gebührend Beachtung finden. Setzen Sie die Höhe des `SemanticZoom`-Controls in der Datei *home.css* für die Portrait-Ansicht auf 100 %:

```
@media screen and (-ms-view-state: fullscreen-portrait) {
    .homepage section[role=main] {
        margin-left: 100px;
    }
    #meinSemanticZoom {
        height: 100%;
    }
}
```

Listing 8.23: ***Erweiterung der Datei*** home.css

Die *Eierlegende Wollmilchsau* ist mit dem bisschen Code, der bislang implementiert wurde, schon recht weit gekommen – nun werden auch verschiedene Ansichten unterstützt. Das nächste Kapitel befasst sich mit den »Contracts«, also den verschiedenen Verträgen, die eine Integration der eigenen App in das Betriebssystem erlauben.