# Einleitung

Super! Die Idee für eine neue tolle Windows 8 App ist gefunden und somit ein wichtiger Schritt schon mal getan. Bevor Sie sich nun aber Hals über Kopf in die Entwicklung stürzen, sollte Sie noch einmal kurz überlegen, was Sie von einer App erwarten, damit Sie eine ihrer Lieblings-Apps wird. Natürlich soll sie ihren Zweck erfüllen, also genau die von Ihnen benötigen UseCases abdecken. Aber damit allein ist es nicht getan. Damit es wirklich Spaß macht mit einer App zu arbeiten, muss sie auch schnell starten, sofort reagieren, nie abstürzen, intuitiv bedienbar sein, gut aussehen, immer funktionieren, usw. Nicht ganz einfach, oder? Selbst Apps der großen Social Media Plattformen können diese Erwartungen nicht immer erfüllen und werden mit schlechten Bewertungen bestraft. Vielleicht kennen Sie das ja auch schon: Sie sitzen gerade im Zug und wollen sich die Neuigkeiten aus Ihrem Bekanntenkreis anschauen, müssen aber gefühlte Stunden warten, bis überhaupt etwas auf dem Bildschirm angezeigt wird – wenn nicht gerade noch ein Tunnel dazwischen kommt. Große Anbieter können sich so etwas vielleicht leisten, sofern Sie aber gerade erst in den Markt einsteigen wollen, muss der erste Wurf sitzen. Also verabschieden wir uns von der idealen Welt, in der wir immer online sind und Webdienste nie ausfallen und kümmern uns um die Offlinefähigkeit Ihrer Anwendung. Die Frage „Wie stelle ich eigentlich fest, ob ich offline bin?“ wird im Kasten „Offlinetypen“ behandelt.

# Methodisches Vorgehen

Wie bei jeder guten Anwendung steht am Anfang der Entwicklung das Requirements- und Usability-Engineering. Auch im Bereich der Offlinefähigkeit sind diese Disziplinen unabdingbar. Sie sollten genau überlegen, welche UseCases offline für Ihre App überhaupt Sinn haben und was im Offline-Modus angezeigt werden soll. Können Sie entscheiden, welche Informationen offline relevant sind oder sollte die Entscheidung dem Anwender überlassen werden? Eine App, die zum Beispiel auf StackOverflow Suchen durchführt, kann offline nicht den vollen Funktionsumfang bieten. Es wäre aber vorstellbar, schon früher gefundene und evtl. markierte Diskussionen (Anwenderentscheidung) offline zu speichern und diese dem Nutzer zu präsentieren. Wahrscheinlich wären für diesen Anwendungsfall auch sehr unterschiedliche Views für den Online- und Offline-Modus notwendig, so dass auch geklärt werden muss, wann man zwischen den Modi hin und her wechselt. Kann dies automatisch geschehen? Im Zweifelsfall sollten Sie dem Anwender die Kontrolle überlassen, denn nichts nervt mehr, als wenn die App ständig automatisch ihre Views umschaltet. Sollte Ihre App für beide Modi dieselben Views verwenden, ist es dennoch wichtig über den Status der Internetverbindung zu informieren. Sie wollen natürlich wissen, ob Sie gerade über die aktuellsten oder nur ältere Daten verfügen. Generell eignet sich jedoch Content, welcher wenigen Änderungen unterworfen ist, gut für die Offline-Verwaltung.

Als weiteres Beispiel – auch für den Lernprozess den man bei der Entwicklung einer guten App durläuft – soll hier die Xing-App für Windows Phone 7 herangezogen werden. Der erste Entwurf sah vor, dass viele Entitäten, wie Kontakte, Nachrichten oder Neuigkeiten aus dem Sozialen Netzwerk zwar offline gespeichert werden, dies aber nur für schon betrachtete Einträge. Bei internen Tests zeigte sich schnell, dass für es für die Anwender vollkommen intransparent und frustrierend war, wenn einzelne Kontakte offline zur Verfügung standen, andere aber nicht. Die Lösung war möglichst alle Entitäten offline zu speichern. Nur zu welchem Zeitpunkt muss dies geschehen? Sollten alle Kontakte geladen und gespeichert werden, wenn man das erste Mal die Kontaktübersicht öffnet? Das Ergebnis wäre eine recht lange Wartezeit, was bei dem Anwender wiederum für Unmut sorgt. Am ehesten wird die Wartezeit beim ersten Start der Anwendung akzeptiert. Jedoch auch zu diesem Zeitpunkt muss der Anwender informiert werden, was die App gerade tut. In der Xing-App gibt es dafür einen Lade-Bildschirm, welcher genau darüber informiert, was gerade geladen wird und wie weit der Prozess schon fortgeschritten ist. (Abbildung 1)

Sind erst einmal alle Daten geladen, ergibt sich ein weiterer Vorteil dieser Vorgehensweise. Beim zweiten Start der Anwendung sind die meisten Daten schon lokal abgelegt und können sehr schnell zugegriffen werden. Die Anwendung startet bedeutend schneller und kann sofort Inhalte anzeigen. Gleichzeitig kann asynchron online nach den neuesten Daten gesucht werden, welche nachträglich den Anwender präsentiert werden. Die Architektur dieses „OfflineFirst“-Ansatzes wird im nächsten Abschnitt genauer erläutert.

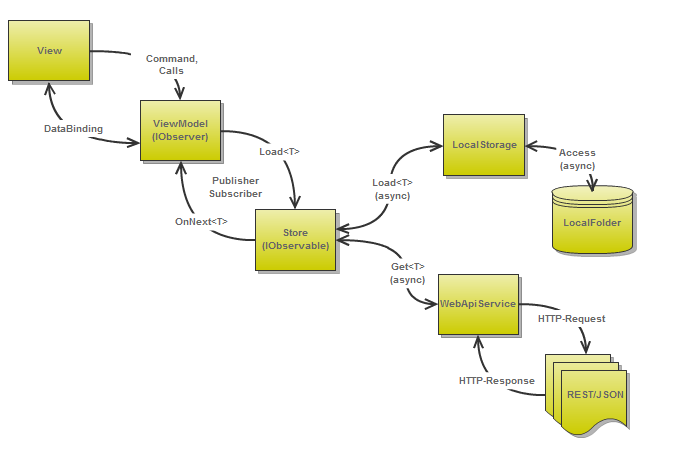
Als Hilfestellung für Ihre App hier noch mal die wichtigsten Frage, die Sie sich stellen sollten:

* Was soll offline gespeichert werden? Geschieht dies automatisch oder wird Content explizit durch den Benutzer gespeichert.
* Muss es unterschiedliche Views für den Online/Offline-Fall geben?
* Wie zeige ich den Online-Status an?
* Wie wechselt man den Online/Offline-Modus? Automatisch oder explizit durch den Benutzer?

# Eine mögliche OfflineFirst Architektur

Im folgenden Abschnitt soll ein möglicher Architekturansatz für eine OfflineFirst Strategie aufgezeigt werden. Die zu dem Artikel mitgelieferte Beispielanwendung liest über einen WebService die aus der guten alten Northwind Datenbank bekannten Customer und ihre Orders ein. Dabei werden für Customers und Orders ein jeweils eigener Store verwendet.

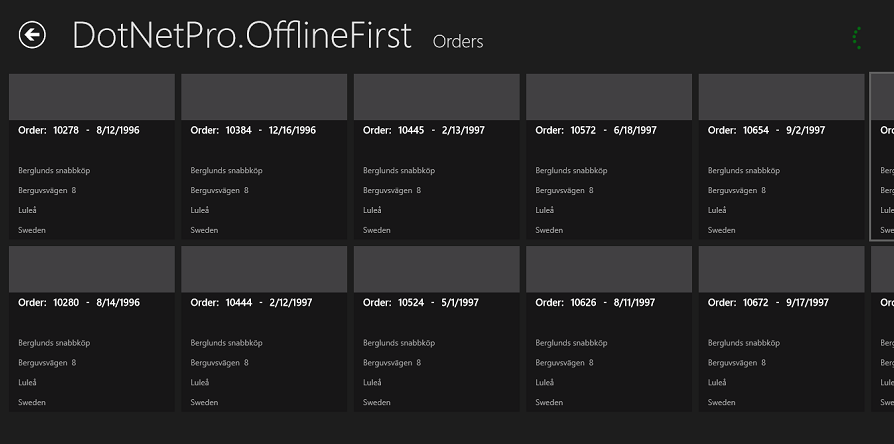
Die dabei verwendeten Klassen werden in Abbildung 2 abgebildet.



Die Beispielarchitektur ist nach dem für WPF Anwendungen allgemein empfohlenen MVVM Pattern umgesetzt. Die Store-Klasse übernimmt hierbei die Rolle des Models. Die ViewModel Klasse selbst registriert sich als Observer über das von Microsoft in .NET4 eingeführt IObservable Interface für alle neuen Daten, die vom Store gelesen werden. Dabei ist es für das ViewModel irrelevant, ob die Daten offline oder online geladen werden. Das ViewModel bekommt alle Daten durch den Aufruf von IObserver.MoveNext geliefert.

Die Klasse Store selbst fragt beim ersten Zugriff zuerst die möglicherweise schon offline gespeicherten Daten ab und liefert diese per Aufruf von OnNext erst einmal aus, bevor der Zugriff auf den WebApi Service erfolgt.

Der Benutzer sieht dadurch die zuletzt gelesenen Daten relativ schnell und muss nicht warten, bis die Daten über den möglicherweise langsamen WebService geladen werden. Wie in Abbildung 3 zu sehen ist, werden die offline geladenen Daten schon angezeigt. Dem Benutzer wird aber durch die Einblendung eines im WinRT Framework ProgressRing genannten Steuerelements suggeriert, dass Daten geladen werden.



## Anwendungsdaten lokal speichern

Es gilt nun zu überlegen, wann diese Daten lokal gespeichert werden sollten. Die Entscheidung kann je nach Anwendungsfall anders ausfallen. Es ist möglich die Daten sofort lokal zu persistieren, oder aber die Daten erst dann lokal zu speichern, wenn die Anwendung in den Ruhezustand geschickt oder sogar beendet wird. Windows 8 Apps unterliegen einem etwas anderen Handlings als man dies von herkömmlichen Windows Anwendungsprogrammen gewohnt ist. In dem Moment, in dem der Benutzer zwischen zwei Windows 8 Apps wechselt, wird die vorher aktive App in den Supended Modus geschickt. Dabei bleiben zwar alle Daten im Speicher, aber der Entwickler bekommt diesen Vorgang durch das Abonnieren des Application.Suspending Ereignisses mit und kann entsprechend darauf reagieren. In Listing 1 wird gezeigt, wie dies in der Beispielapplikation umgesetzt ist.

private async void OnSuspending(object sender, SuspendingEventArgs e)

{

var deferral = e.SuspendingOperation.GetDeferral();

await SuspensionManager.SaveAsync();

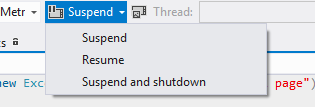
await Container.Resolve<ApplicationStore>().SaveAsync();

deferral.Complete();

}

In den Projektvorlagen die Microsoft für die Erstellung von Windows 8 Applikationen mitliefert, ist dieser Eventhandler schon enthalten. Hinzugekommen ist nur die Zeile, in der ApplicationStore.SaveAsync Methode aufgerufen wird. Zu der Aufgabe des SupensionManagers kommen wir noch später in diesem Artikel. Wichtig zu wissen ist hier noch, die Bedeutung des Deferral Objektes. Da jetzt die SaveAsync-Methoden, wie schon der Name suggeriert, Asynchron ausgeführt werden, wartet die Runtime normalerweise nicht, bis die Methoden wirklich ausgeführt wurden, sondern das Programm könnte unter Umständen beendet werden, bevor die Speichervorgänge durchgeführt wurden. Um dies zu verhindern, fordert man über die SuspendingEventArgs ein SupendingDeferral Objekt an, das man erst nach Ausführung der beiden SaveAsync Methoden auf Complete setzt. Die WinRT Runtime weiß dadurch, dass hier erst asynchrone Methoden beendet werden müssen, bis die Applikation wirklich beendet werden kann. Unter [1] ist in der MSDN Dokumentation genauer erklärt, was bei Suspend und dem entsprechenden Gegenstück Resume zu beachten ist.

Um das Suspend Ereignis und die entsprechende Ereignisbehandlung selbst in der Entwicklungsumgebung testen zu können, reicht es nicht, im Simulator von der zu testenden App zu einer anderen zu wechseln. Visual Studio stellt hierfür über die Toolbar Debug Location die in Abbildung 4 angezeigten Befehle zur Verfügung, um die Ereignisses explizit auszulösen.



Bevor wir uns anschauen, wo und wie wir nun die über den in der Beispielapplikation verwendeten WebService geladenen Daten speichern können sollten ein paar grundlegende Aspekte über die Möglichkeiten in Windows 8 Applikationen Daten lokal zu speichern erläutert werden.

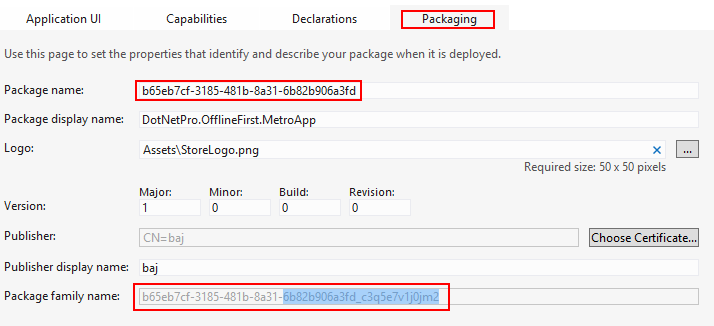
Ähnlich wie in Silverlight oder Windows Phone Anwendungen kann man auch in Windows 8 Anwendungen seine nicht an jeder beliebigen Stelle im Dateisystem speichern oder lesen.

Windows 8 Anwendungen werden in einer Art Sandbox (App Container) ausgeführt. Jede App hat einen eigenen abgegrenzten Speicherbereich in den Anwendungsdaten des aktuellen Benutzerkontos. Das aus Silverlight als IsolatedStorage bekannte Gegenstück nennt sich unter Windows 8 nennt sich LocalFolder und lässt sich innerhalb einer Windows 8 Anwendung über ApplicationData.Current.LocalFolder abfragen.

Wenn wir uns diesen Ordern abfragen, bekommen wir in der Beispielanwendung folgenden Verzeichnisnamen zurückgeliefert:

%LocalAppData%\Packages\**b65eb7cf-3185-481b-8a31-6b82b906a3fd\_c3q5e7v1j0jm2**\LocalState

Die lange Zeichenkette im Verzeichnisnamen ergibt sich aus dem Package Namen, den man im Application Manifest festlegen kann, wie dies in der Abbildung 5 zu sehen ist.



In diesem Verzeichnis haben wir nun die erforderlichen Lese- und natürlich auch Schreibrechte.

In Listings 2 wird nun gezeigt, wie wir die übergebenen Daten als Xml-Datei unter zu Verwendung des DataContractSerializers serialisieren und speichern können.

public async Task SaveAsync<T>(string key, T objectToSerialize, List<Type> knownTypes)

{

var localFolder = ApplicationData.Current.LocalFolder;

StorageFile file = await localFolder.CreateFileAsync(key + ".xml", CreationCollisionOption.ReplaceExisting);

IRandomAccessStream raStream = await file.OpenAsync(FileAccessMode.ReadWrite);

using (IOutputStream outStream = raStream.GetOutputStreamAt(0))

{

var serializer = new DataContractSerializer(objectToSerialize.GetType(), knownTypes);

serializer.WriteObject(outStream.AsStreamForWrite(), objectToSerialize);

await outStream.FlushAsync();

}

raStream.Dispose();

}

Zu beachten ist hierbei natürlich, das die Daten als Xml-Datei unverschlüsselt und im Klartext lesbar abgespeichert werden. Obwohl das Verzeichnis für unsere Beispielanwendung explizit als Datenverzeichnis zur Verfügung gestellt wurde und andere Windows 8 Apps auf diesen Ordern nicht ohne weiteres zugreifen können, ist dieses Verzeichnis und damit die darin enthaltenen Dateien natürlich auf herkömmlichen Wege für den aktuellen Benutzer oder native Windows Programme lesbar. Sollten die Daten sicherheitskritisch oder nur nach Authentifizierung im Programm lesbar sein, sollten sie entsprechend verschlüsselt werden.

Sollten für den Zugriff auf einen WebService spezielle Credentials erforderlich sein, so stellt die WinRT Runtime zu Lesen und Speichern dieser Credentials die Klasse PasswordCredential aus den Namensbereich Windows.Security.Credentials zur Verfügung. Diese Credentials werden allerdings nicht im lokalen Anwendungsbereich gespeichert, sondern in einem eigenen Roaming-Ordner. Unter [2] ist in der MSDN Dokumentation nachzulesen, welche Möglichkeiten für Windows 8 Anwendungen zur Verfügung stehen, um UserCredentials zu speichern und zu lesen.

Über die Roaming Ordner lassen sich im Übrigen weitere Daten abspeichern, die dann zwischen mehreren Windows 8 Geräten benutzerbezogen synchronisiert werden können. Prädestiniert für solche Daten sind anwendungsspezifische Benutzereinstellungen, die der Benutzer möglicherweise nicht auf jedem seiner Geräte nicht von neuem vornehmen möchte.

Der Roaming Order lässt sich über ApplicationData.Current.RoamingFolder abfragen. Welche Empfehlungen Microsoft zu, Umgang mit Roaming Daten hat, ist unter [3] nachzulesen. Ein wichtiger und zu beachtender Aspekt hierbei ist, dass die Größe des zur Synchronisation verfügenden Cloud-Speichers begrenzt ist. Diese kann über ApplicationData.Current.RoamingStorageQuota abgefragt werden.

## Lokale Anwendungsdaten lesen

Nachdem wir nun wissen, wie wir die Anwendungsdaten lokal speichern können, sollten wir uns nun natürlich auch Gedanken machen, wann und wie wir wieder aus diese Daten zugreifen können und wie wir diese Daten mit den vom WebService abgefragten aktuelleren Daten synchronisieren können. Je nach Anwendungsfall kann hierfür eine ganz unterschiedliche Strategie erforderlich sein.

In unserer Beispielanwendung beschränken wir uns darauf, die im jeweiligen Store gespeicherten Daten einfach durch die vom Webservice zurückgelieferten Daten zu ersetzen.

In Listing 3 wird exemplarisch gezeigt, wie der CustomerStore zuerst den OfflineStore abfragt um, dem Benutzer möglichst schnell die zuletzt lokal abgespeicherten Customers anzuzeigen, bevor die Daten dann explizit über den WebApiService online abgefragt werden.

public async Task LoadAsync()

{

if (\_customers == null)

{

\_customers = await this.OfflineStore.LoadAsync<List<Customer>>("customers", this.KnownTypes);

this.OnNext(\_customers);

}

if (this.NetworkStatus.IsOnline)

{

var customers = new List<Customer>(await this.WebApiService.GetCustomersAsync());

this.OnNext(customers);

\_customers.Clear();

\_customers.AddRange(customers);

}

}

Durch den Aufruf von IObservable.OnNext wird hierbei dem jeweiligen ViewModel, das die Rolle des Observers übernimmt, die jeweiligen Daten zur Verfügung gestellt. Das ViewModel selbst muss in erkennen, dass es die Daten zu den gleichen Customern möglicherweise zwei Mal bekommt.

In Listing 4 ist eine mögliche Implementierung abgebildet:

private void OnNext(IEnumerable<Customer> customers)

{

Dispatch.Action(() => CreateOrUpdateViewModels(customers));

}

private void CreateOrUpdateViewModels(IEnumerable<Customer> customers)

{

foreach (var customer in customers)

{

string customerId = customer.CustomerId;

var viewModel = (from vm in this.Customers where vm.CustomerId == customerId select vm).FirstOrDefault();

if (viewModel == null)

{

viewModel = new CustomerViewModel() { Customer = customer };

this.Customers.Add(viewModel);

}

else

{

viewModel.Customer = customer;

}

}

}

In Listing 3 können wir auch sehen, dass der Zugriff auf den WebService nur für den Fall durchgeführt wird, in dem auch eine Verbindung zum Internet und damit natürlich auch dem WebService besteht.

Um das feststellen zu können, stellt die Window RT Api stellt hierfür über die Klasse NetworkInformation die Methode GetInternetConnectionProfile zur Verfügung. Im Online Zustand gibt diese Methode ein entsprechendes Connection-Objekt zurück. Über dieses Profil kann dann über GetNetworkConnectionLevel abgefragt werden, ob aktuell eine Verbindung zum Internet besteht.

In Listing 4 wird die entsprechende Implementierung von NetworkStatus.IsOnline gezeigt.

public bool IsOnline

{

get

{

return (NetworkInformation.GetInternetConnectionProfile() != null &&

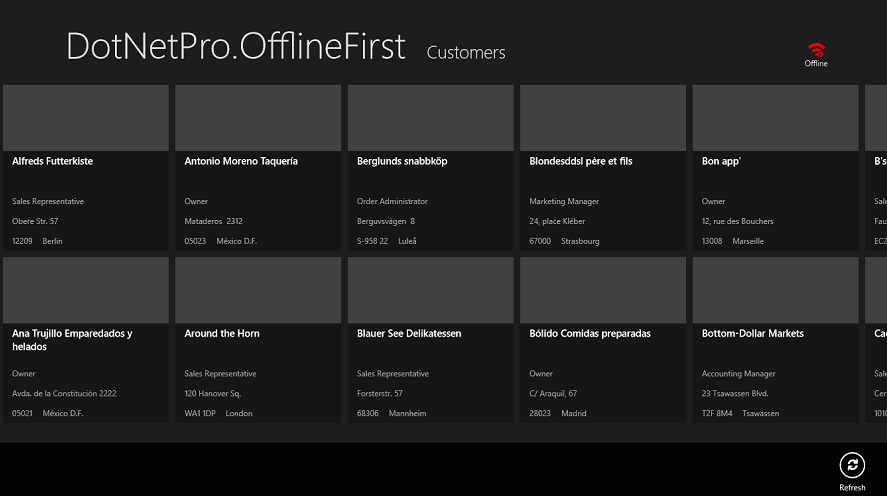
NetworkInformation.GetInternetConnectionProfile().GetNetworkConnectivityLevel() == NetworkConnectivityLevel.InternetAccess);

}

}

Über die aktuelle Geschwindigkeit und Qualität der Internetverbindung und ob der WebService selbst gerade online ist, kann die NetworkInformation Klasse natürlich keine Auskunft geben. Dieser Zustand muss separat behandelt werden.

Da die NetworkInformation Klasse über das Ereignis NetworkStatusChanged relativ zeitnah mitteilt, sobald sich der Status geändert hat, kann darüber dem Benutzer wie in Abbildung 5 zu sehen ist, visuell angezeigt werden, ob gerade eine Internetverbindung besteht.



# Fazit

Durch die Möglichkeit Windows 8 oder Windows RT nicht nur auf lokalen Rechnern oder Notebooks, sondern auch auf mobilen Geräten wie Tablets oder gar Smartphones zu verwenden und diese Geräte zwangsläufig nicht immer Online sind, gilt es, sich Strategien zu überlegen, wie man mit diesem Offline-Zustand umzugehen hat und dem Benutzer trotzdem keine leeren Anwendungsoberfläche ohne Daten anzeigen will.

Über die in der Windows 8 API zur Verfügung gestellten Möglichkeiten ist es dabei möglich, diese Daten in lokalen oder Roaming Ordnern zu speichern und dem Benutzer für einen späteren Offlinezugriff zur Verfügung zu stellen. Im Sinne des Offlinefirst Ansatzes werden die lokal gespeicherten Daten aber eben nicht nur für den Offline Modus genutzt, sondern auch um dem Benutzer möglichst schnell die aus der letzten Online-Sitzung gespeicherten Daten anzuzeigen, bevor die aktuellsten Daten über eine möglichweise langsame Datenverbindung abgefragt werden.

[1] <http://msdn.microsoft.com/en-us/library/windows/apps/hh465088.aspx>

[2] <http://msdn.microsoft.com/en-us/library/windows/apps/xaml/br229572.aspx>

[3] <http://msdn.microsoft.com/en-us/library/windows/apps/hh465094.aspx>