http://msdn.microsoft.com/ja-jp/library/windows/apps/br211380
MSDN: デベロッパー センター - Metro スタイル アプリ>ドキュメント>はじめに>C# と Visual Basic の選択

※ 2012/8/9 取得
※ 2012/8/10 誤記訂正等 biac

パート 3: ブログ リーダーを作成する (C# と XAML を使った Metro スタイル アプリ)

この記事の内容

[このドキュメントは暫定版であり、変更されることがあります。]

ここでは、C# を使って Metro スタイル アプリを作るために必要な基本的なコードと概念について紹介します。Extensible Application Markup Language (XAML) を使って UI を定義し、選んだ言語を使ってアプリのロジックを作ります。

別のプログラミング言語を使った方法については、次のトピックをご覧ください。

- JavaScript を使った初めての Metro スタイル アプリの作成
- C++ を使った初めての Metro スタイル アプリの作成

ロードマップ: このトピックと他のトピックとの関連については、「C# または Visual Basic を使った Metro スタイル アプリのためのロードマップ」をご覧ください。

目標

このチュートリアルでは、Metro スタイル アプリの作成に使う機能を簡単に紹介します。単純なブログ リーダー アプリを作成するプロセスに沿って、レイアウト、コントロール、テンプレート、データ バインドなどの、XAML を使った開発の中心となる概念を紹介します。Microsoft Visual Studio Express 2012 RC for Windows 8 に付属するページ テンプレートとナビゲーションを使って、手軽にアプリ開発に着手する方法を説明します。次に、カスタム スタイルを使ってアプリの外観を変更する方法と、UI をさまざまなレイアウトやビューに適合させる方法を説明します。最後に、作成したアプリを Windows 8 Release Preview と統合し、Windows ストアに公開する方法について簡単に説明します。このチュートリアルを終えると、自分で Metro スタイル アプリを構築する準備が整います。

このチュートリアルは読むのに約 30 分かかります。演習を行う場合はそれ以上の時間がかかります。

Hello World

C# または Visual Basic を使って Metro スタイル アプリを作る場合、通常は XAML を使って UI を定義し、関連する分離コード ファイルのアプリ ロジックについては自分で選んだ言語で作ります。C# または Visual Basic を使った Metro スタイル アプリの XAML UI フレームワークは、Windows ランタイム(Windows RT)の Windows.UI.Xaml.* 名前空間に含まれています。Windows Presentation Foundation (WPF)、Microsoft Silverlight、または Silverlight for Windows Phone を使ってアプリを作った経験がある場合、これは馴染みのあるプログラミング モデルであるため、この経験を活かして C++、C#、Visual Basic を使った Metro スタイル アプリを作ることができます。

次の例は、単純な Hello World アプリの UI を定義する XAML とその分離コード ページを示しています。この例は単純ですが、部分クラス、レイアウト、コントロール、プロパティ、イベントなど、XAML ベースのプログラミング モデルに重要ないくつかの概念を含んでいます。

XAML

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

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <StackPanel>
            <Button Content="Click Me" Click="HelloButton_Click" />
            <TextBlock x:Name="DisplayText" FontSize="48" />
        </StackPanel>
    </Grid>
</Page>
```

C#

```
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;

namespace WindowsBlogReader
{
    public sealed partial class MainPage : Page
    {
        public MainPage()
```

```
        {
            InitializeComponent();
        }

        private void HelloButton_Click(object sender, RoutedEventArgs e)
        {
            DisplayText.Text = "Hello World";
        }
    }
}
```

Hello World アプリは、最初のアプリとしては手頃なアプリです。しかし、この段階から勉強を始めるとアプリで収益を上げるまでにかなりの時間がかかりますから、ここからは Windows 8 用の Metro スタイル アプリを作成してみましょう。手始めに、RSS (Really Simple Syndication) 2.0 フィードまたは Atom 1.0 フィードからデータをダウンロードして表示する、単純なブログ リーダーをサンプル アプリとして作ります。ここでは、Windows チーム ブログ サイトからのフィードを取得するようにします。「Windows アプリの開発」ブログは、Windows チーム ブログの一例です。

Visual Studio での Windows Metro スタイル アプリの作成

このセクションの内容:

- Visual Studio Express 2012 RC for Windows 8 で新しい **Windows Metro スタイル** プロジェクトを作る方法

Microsoft Visual Studio は、Windows アプリを開発するための強力な統合開発環境 (IDE) です。Visual Studio は、ソース ファイル管理機能、ビルド、展開、起動の統合サポート機能、XAML、Visual Basic、C#、C++、グラフィックス、マニフェストの編集機能、デバッグ機能などを備えています。Visual Studio にはいくつかのエディションがありますが、ここでは Visual Studio Express 2012 RC for Windows 8 を使います。このエディションは Metro スタイル アプリ用 Windows ソフトウェア開発キット (Windows SDK) と共に無料でダウンロードできるので、Metro スタイル アプリの構築、パッケージ化、展開に必要なツールを揃えることができます。

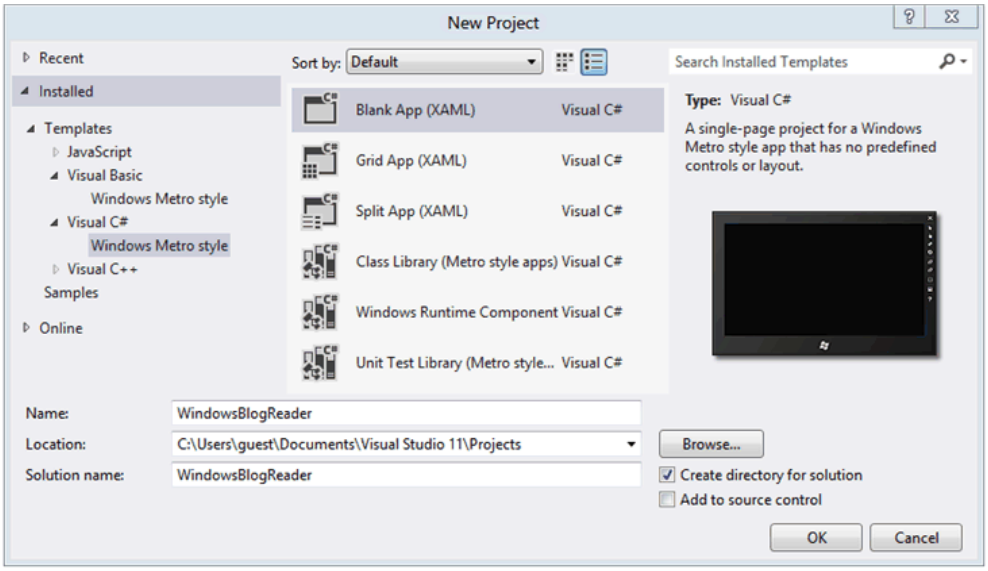
アプリの作成を開始するには、C# または Visual Basic を使って新しい **Windows Metro スタイル** プロジェクトを作ります。Visual Studio Express 2012 RC for Windows 8 には、**Windows Metro スタイル** プロジェクトのいくつかのテンプレートが含まれているため、さまざまなレイアウトを使ってアプリを簡単に作ることができます。**[Blank App (XAML)]**~~**(新しいアプリ (XAML))**~~ プロジェクト テンプレートには、すべての **Windows Metro スタイル** アプリに必要な最小限のファイルが用意されています。

Visual Studio Express 2012 RC for Windows 8 について詳しくは、「Visual Studio 11 Beta」をご覧ください。

新しい Windows Metro スタイル プロジェクトを作るには

1. Visual Studio Express 2012 RC for Windows 8 を開きます。
2. **[ファイル]** の [新規作成] の **[**~~**新しい**~~**プロジェクト...]** をクリックします。**[新しいプロジェクト]** ダイアログ ボックスが表示されます。
3. **[インストール済み]** ウィンドウで、**[Visual C#]** または **[Visual Basic]** を展開します。
4. テンプレートの種類として **[Windows Metro** ~~**スタイル**~~ **Style]** を選びます。
5. 中央のウィンドウで、**[Blank App (XAML)]**~~**(新しいアプリ (XAML))**~~ を選びます。
6. プロジェクトの名前を入力します。このプロジェクトに "WindowsBlogReader" という名前を付けます。(必要に応じて、保存する場所(フォルダー)も変更してください。)

   次の図では、Visual Studio Express 2012 RC for Windows 8 で新しいプロジェクトを作っています。



7. **[OK]** をクリックします。プロジェクト ファイルが作られます。

プロジェクトを作ると、Visual Studio によってプロジェクト ファイルが作られ、**ソリューション エクスプローラー**に表示されます。**[Blank App (XAML)]**~~**(新しいアプリ (XAML))**~~ テンプレートによって作られるファイルを次の表に示します。

ファイル名	説明
Properties/AssemblyInfo (.vb or .cs)	生成されたアセンブリに埋め込まれる名前とバージョンのメタデータが格納されます。
Package.appxmanifest	表示名、説明、ロゴ、機能など、アプリを説明するメタデータが格納されます。
Assets/*	ロゴとスプラッシュ画面用の既定のイメージ ファイルです。これらのファイルは、自分で用意したファイルで置き換えることができます。
Common/StandardStyles.xaml	アプリの既定のスタイルとテンプレートが格納されます。
App.xaml、App.xaml.* (.vb, .cs)	これらのファイルでは、アプリレベルのロジックを指定します。App クラスは、ユーザー インターフェイスを表示するために必要です。
MainPage.xaml	ユーザー インターフェイスを作るために使う既定のスタート ページです。
MainPage.xaml.* (.vb, .cs)	既定のスタート ページのロジックが格納された分離コ

	ード ファイルです。

これらのファイルとテンプレートについて詳しくは、「テンプレートを使った Metro スタイル アプリ開発に着手する (C#, C++, Visual Basic)」をご覧ください。

アプリ機能の指定

このセクションの内容:

- 機能を使う方法
- アプリケーション マニフェスト デザイナーでアプリ機能を指定する方法

Metro スタイル アプリは、ファイル システム、ネットワーク リソース、ハードウェアへのアクセスが制限されるセキュリティ コンテナー内で実行されます。 ユーザーが Windows ストアからアプリをインストールするたびに、Package.appxmanifest ファイル内のメタデータに基づいて、アプリの動作に必要な機能が判定されます。たとえば、インターネット上のデータ、ユーザーのドキュメント ライブラリ内のドキュメント、またはユーザーの Web カメラやマイクにアクセスする必要があるアプリがあるとします。このアプリをインストールすると、必要な機能がユーザーに対して表示されます。このアプリがこれらのリソースにアクセスできるようにするには、ユーザーがアクセス許可を与える必要があります。必要なリソースへのアクセス許可をアプリが要求せず、その許可が得られない場合、実行時にアプリに対してそのリソースへのアクセスは許可されません。

※ ここで言う「機能」の原語は function ではなく、capability(～をする能力)。

書式変更: 蛍光ペン

いくつかの一般的な機能を次に示します。

機能	説明
ドキュメント ライブラリ アクセス	アプリがユーザーのドキュメント ライブラリにアクセスしてファイルを追加、変更、または削除できるようにします。アプリは、マニフェストに宣言されている種類のファイルにのみアクセスできます。HomeGroup コンピューターのドキュメント ライブラリにはアクセスできません。
エンタープライズ認証	ドメイン資格情報を必要とするイントラネット リソースにアプリが接続できるようにします。
ホームまたは社内ネットワーク	入力方向のアクセスと、アプリからユーザーの信頼済みネットワーク (ホーム ネットワーク、エンタープライズ ネットワークなど) への出力方向のアクセスを許可します。重要なポートへの入力方向のアクセスは常にブロックされます。
インターネット (クライアントとサーバー)	アプリがインターネットやパブリック ネットワークにアクセスできるようにし、インターネットからアプリへの着信接続を許可します。重要なポートへの入力方向のアクセスは常にブロックされます。これは、インターネット (クライアント) 機能のスーパーセットです。両方の機能を宣言する必要はありません。
インターネット (クライアント)	アプリがインターネットやパブリック ネットワークにアクセスできるようにします。インターネットにアクセスする必要があるほとんどのアプリは、この機

	能を使います。
場所	アプリがユーザーの現在の場所にアクセスできるようにします。 ※ GPS から得られる緯度・経度情報
マイク	アプリがユーザーのマイクにアクセスできるようにします。
音楽ライブラリ	アプリがユーザーの音楽ライブラリにアクセスしてファイルを追加、変更、または削除できるようにします。さらに、HomeGroup コンピューター上の音楽ライブラリや、ローカル接続されているメディア サーバー上の各種の音楽ファイルへのアクセスも許可されます。
画像ライブラリ アクセス	アプリがユーザーの画像ライブラリにアクセスしてファイルを追加、変更、または削除できるようにします。さらに、HomeGroup コンピューター上の画像ライブラリや、ローカル接続されているメディア サーバー上の各種の画像ファイルへのアクセスも許可されます。
近接通信	アプリがユーザーの近距離通信 (NFC) デバイスにアクセスできるようにします。
リムーバブル記憶域	アプリがリムーバブル記憶域装置 (外部ハード ドライブ、USB フラッシュ ドライブなど) にアクセスしてファイルを追加、変更、または削除できるようにします。アプリは、マニフェストに宣言されている種類のファイルにのみアクセスできます。HomeGroup コンピューター上のリムーバブル記憶域装置にはアクセスできません。
共有ユーザー証明書	ソフトウェア証明書とハードウェア証明書 (スマート カード証明書など) にアプリがアクセスできるようにします。
テキスト メッセージング	アプリがテキスト メッセージング機能にアクセスできるようにします。
ビデオ ライブラリ アクセス	アプリがユーザーのビデオ ライブラリにアクセスしてファイルを追加、変更、または削除できるようにします。さらに、HomeGroup コンピューター上のビデオ ライブラリや、ローカル接続されているメディア サーバー上の各種のビデオファイルへのアクセスも許可されます。
Web カメラ	アプリがユーザーのカメラにアクセスできるようにします。

アプリに機能を追加するには

1. **ソリューション エクスプローラー**で、Package.appxmanifest をダブルクリックします。**アプリケーション マニフェスト デザイナー**でファイルが開かれます。
2. アプリケーション マニフェスト デザイナーで、**[機能]** タブをクリックします。

3. アプリに必要なそれぞれの機能の横のチェック ボックスをオンにします ([インターネット (クライアント)] は既定でオンになっています。ブログ リーダー アプリでは、この設定が必須です。)
4. ファイルを保存して閉じます。

アプリでのデータの取得

このセクションの内容:

- カスタム データ クラスを作る方法
- RSS または Atom データ フィードを非同期的に取得する方法

インターネット機能を宣言したので、アプリでブログ フィードを取得するためのコードを記述します。Windows チーム ブログでは、投稿の全文が RSS と Atom の両方の形式で公開されています。リーダー アプリに表示するブログ データは、ブログへの最新の投稿のタイトル、作成者、日付、内容です。

最初に、それぞれの投稿のデータをダウンロードする必要があります。Windows ランタイムには、このフィード データ処理の多くを自動的に実行するクラスが複数用意されています。これらのクラスは、**Windows.Web.Syndication** 名前空間に含まれています。これらのクラスを直接使ってデータを UI に表示することもできますが、このブログ リーダーでは独自にデータ クラスを作ります。これにより、柔軟性が高まり、RSS フィードと Atom フィードを同じ方法で扱うことができるようになります。

▶プロジェクトに新しいクラス ファイルを追加するには

1. **[プロジェクト]** の **[クラスの追加]** をクリックします。**[新しい項目の追加]** ダイアログ ボックスが表示されます。
2. クラス ファイルの名前を入力します。この例では **FeedData** を使います。
3. **[追加]** をクリックします。これで、新しいクラス ファイルが作成されます。

ここでは、ブログ リーダー アプリでフィード データを取得して保持するために、3 つのクラスを使います。これらの 3 つのクラスすべてを、**FeedData** (.cs または .vb) という名前の 1 つのファイルに格納します。**FeedData** クラスは、RSS フィードまたは Atom フィードに関する情報を保持します。**FeedItem** クラスは、フィードに含まれる個々のブログ投稿に関する情報を保持します。**FeedDataSource** クラスには、フィードのコレクションと、ネットワークからフィードを取得するためのメソッドが含まれます。これらのクラスのコードを次に示します。
※ 以下のコードを、作成した FeedData.cs にコピー&ペーストしてください。

コメント [Y1]: 実際の製品開発では、ちゃんと別々のファイルにすべき。

C#

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Threading.Tasks;
using Windows.Web.Syndication;
```

```
namespace WindowsBlogReader
{
    // FeedData
    // Holds info for a single blog feed, including a list of blog posts (FeedItem)
    public class FeedData
    {
        public string Title { get; set; }
        public string Description { get; set; }
        public DateTime PubDate { get; set; }

        private List<FeedItem> _Items = new List<FeedItem>();
        public List<FeedItem> Items
        {
            get
            {
                return this._Items;
            }
        }
    }

    // FeedItem
    // Holds info for a single blog post
    public class FeedItem
    {
        public string Title { get; set; }
        public string Author { get; set; }
        public string Content { get; set; }
        public DateTime PubDate { get; set; }
        public Uri Link { get; set; }
    }

    // FeedDataSource
    // Holds a collection of blog feeds (FeedData), and contains methods needed to
    // retreive the feeds.
    public class FeedDataSource
    {
        private ObservableCollection<FeedData> _Feeds
            = new ObservableCollection<FeedData>();
        public ObservableCollection<FeedData> Feeds
        {
            get
            {
                return this._Feeds;
            }
        }

        public async Task GetFeedsAsync()
        {
```

```
            Task<FeedData> feed1 =
GetFeedAsync("http://windowsteamblog.com/windows/b/developers/atom.aspx");
            Task<FeedData> feed2 =
GetFeedAsync("http://windowsteamblog.com/windows/b/windowsexperience/atom.aspx");
            Task<FeedData> feed3 =
GetFeedAsync("http://windowsteamblog.com/windows/b/extremewindows/atom.aspx");
            Task<FeedData> feed4 =
GetFeedAsync("http://windowsteamblog.com/windows/b/business/atom.aspx");
            Task<FeedData> feed5 =
GetFeedAsync("http://windowsteamblog.com/windows/b/bloggingwindows/atom.aspx");
            Task<FeedData> feed6 =
GetFeedAsync("http://windowsteamblog.com/windows/b/windowssecurity/atom.aspx");
            Task<FeedData> feed7 =
GetFeedAsync("http://windowsteamblog.com/windows/b/springboard/atom.aspx");
            Task<FeedData> feed8 =
GetFeedAsync("http://windowsteamblog.com/windows/b/windowshomeserver/atom.aspx");
            // There is no Atom feed for this blog, so we use the RSS feed.
            Task<FeedData> feed9 =
GetFeedAsync("http://windowsteamblog.com/windows_live/b/windowslive/rss.aspx");
            Task<FeedData> feed10 =
GetFeedAsync("http://windowsteamblog.com/windows_live/b/developer/atom.aspx");
            Task<FeedData> feed11 =
                GetFeedAsync("http://windowsteamblog.com/ie/b/ie/atom.aspx");
            Task<FeedData> feed12 =
GetFeedAsync("http://windowsteamblog.com/windows_phone/b/wpdev/atom.aspx");
            Task<FeedData> feed13 =
GetFeedAsync("http://windowsteamblog.com/windows_phone/b/wmdev/atom.aspx");

            this.Feeds.Add(await feed1);
            this.Feeds.Add(await feed2);
            this.Feeds.Add(await feed3);
            this.Feeds.Add(await feed4);
            this.Feeds.Add(await feed5);
            this.Feeds.Add(await feed6);
            this.Feeds.Add(await feed7);
            this.Feeds.Add(await feed8);
            this.Feeds.Add(await feed9);
            this.Feeds.Add(await feed10);
            this.Feeds.Add(await feed11);
            this.Feeds.Add(await feed12);
            this.Feeds.Add(await feed13);
        }

        private async Task<FeedData> GetFeedAsync(string feedUriString)
        {
            // using Windows.Web.Syndication;
            SyndicationClient client = new SyndicationClient();
            Uri feedUri = new Uri(feedUriString);
```

```
            try
            {
                SyndicationFeed feed = await client.RetrieveFeedAsync(feedUri);

                // This code is executed after RetrieveFeedAsync returns the
                // SyndicationFeed.
                // Process it and copy the data we want into our FeedData and
                // FeedItem classes.
                FeedData feedData = new FeedData();

                feedData.Title = feed.Title.Text;
                if (feed.Subtitle != null && feed.Subtitle.Text != null)
                {
                    feedData.Description = feed.Subtitle.Text;
                }
                // Use the date of the latest post as the last updated date.
                feedData.PubDate = feed.Items[0].PublishedDate.DateTime;

                foreach (SyndicationItem item in feed.Items)
                {
                    FeedItem feedItem = new FeedItem();
                    feedItem.Title = item.Title.Text;
                    feedItem.PubDate = item.PublishedDate.DateTime;
                    feedItem.Author = item.Authors[0].Name.ToString();
                    // Handle the differences between RSS and Atom feeds.
                    if (feed.SourceFormat == SyndicationFormat.Atom10)
                    {
                        feedItem.Content = item.Content.Text;
                        feedItem.Link
                            = new Uri("http://windowsteamblog.com" + item.Id);
                    }
                    else if (feed.SourceFormat == SyndicationFormat.Rss20)
                    {
                        feedItem.Content = item.Summary.Text;
                        feedItem.Link = item.Links[0].Uri;
                    }
                    feedData.Items.Add(feedItem);
                }
                return feedData;
            }
            catch (Exception)
            {
                return null;
            }
        }
    }
}
```

これらのデータ クラスを追加したので、**[ビルド]**、**[ソリューションのビルド]** の順にクリックするか、~~F7~~ F6 キーを押して、エラーなしでソリューションがビルドされることを確認します。

コメント [Y2]: Visual Studio の設定によってキーが違う。メニュー [ビルド] - [ソリューションのビルド]

フィード データの取得

ブログ フィードをダウンロードする方法を詳しく見てみましょう。 **Windows.Web.Syndication.SyndicationClient** クラスを使うと、完全に解析された RSS フィードまたは Atom フィードを取得できます。したがって、XML の解析について考えることなくデータを使うことができるようになります。 **SyndicationClient** クラスを使ってフィードをダウンロードするには、非同期の **RetrieveFeedAsync** メソッドを使う必要があります。非同期プログラミング モデルは、Windows ランタイムでアプリの応答性を保つために一般的に使われます。非同期メソッドを使った場合に予想される面倒な処理は、ほとんど対応する必要はありません。

C# と Visual Basic での await の使用

C# や Visual Basic で **await** キーワードを使った場合、フィードを非同期的に取得するためのコードは、フィードを同期的に取得するためのコードと似ています。コードを次に示します。

GetFeedsAsync メソッド内で、取得するブログ フィードごとに GetFeedAsync を呼び出します。このとき、可能であれば Atom フィードの URL を渡します。Atom フィードには、表示する作成者データが含まれるためです。各ブログ フィードが返されるときに、FeedDataSource.Feeds コレクションにこのデータを追加します。

C#

```
public async Task GetFeedsAsync()
{
    Task<FeedData> feed1 =
        GetFeedAsync("http://windowsteamblog.com/windows/b/developers/atom.aspx");
    ...
    this.Feeds.Add(await feed1);
    ...
}
```

それでは、GetFeedAsync メソッドをさらに詳しく確認して、**await** キーワードの使い方を見てみましょう。最初に、**async** キーワードがメソッド シグネチャに追加されている点に注目してください。

C#

```
private async Task<FeedData> GetFeedAsync(string feedUriString)
{
```

書式変更: フォント : 太字, 蛍光ペン

```
...
}
```

await キーワードは、**async** として定義されたメソッドでのみ使うことができます。戻り値の型を **Task<FeedData>** に指定します。この指定によって、メソッドが取得する FeedData オブジェクトを表す **Task** を生成するように、コンパイラに指示します。

メソッドの内部では、**SyndicationClient** をインスタンス化して **RetrieveFeedAsync** メソッドを呼び出し、目的の RSS 情報または Atom 情報を含む **SyndicationFeed** を取得します。

C# (GetFeedAsync()メソッドの 9 行目付近。try ブロックの先頭。)

```
SyndicationFeed feed = await client.RetrieveFeedAsync(feedUri);
```

書式変更: 蛍光ペン

ここでは、**await** キーワードによって、コンパイラがさまざまな作業を自動的に実行するように指示しています。コンパイラは、この呼び出しの後のメソッドの残りの部分を、呼び出しの応答が返されるときに実行されるコールバックとしてスケジュールします。次に、コンパイラは、アプリの応答性を保つために、即座に制御を呼び出し元スレッド（通常は UI スレッド）に返します。このメソッドの最終的な出力を表す **Task**、つまり FeedData オブジェクトが、この時点で呼び出し元に返されます。

RetrieveFeedAsync から目的のデータを保持した **SyndicationFeed** が返されると、メソッドの残りのコードが実行されます。ここで重要なのは、元の呼び出しの実行元と同じスレッド コンテキスト (UI スレッド) でコードが実行されるという点です。したがって、このコードの UI を更新する場合、ディスパッチャーを使った操作は必要ありません。**SyndicationFeed** を取得したら、必要な部分を FeedData データ クラスと FeedItem データ クラスにコピーします。

コメント [Y3]: UI スレッドから呼び出したから、UI スレッド。非同期実行しているメソッドから呼び出した場合には、UI スレッドに戻らないので、注意。

C# (GetFeedAsync()メソッドの十数行目あたりから。先ほどのコードの続き。)

```
// This code is executed after RetrieveFeedAsync returns the SyndicationFeed.
// Process it and copy the data we want into our FeedData and FeedItem classes.
FeedData feedData = new FeedData();

feedData.Title = feed.Title.Text;
if (feed.Subtitle != null && feed.Subtitle.Text != null)
{
    feedData.Description = feed.Subtitle.Text;
}
// Use the date of the latest post as the last updated date.
feedData.PubDate = feed.Items[0].PublishedDate.DateTime;

foreach (SyndicationItem item in feed.Items)
```

```
{
    FeedItem feedItem = new FeedItem();
    feedItem.Title = item.Title.Text;
    feedItem.PubDate = item.PublishedDate.DateTime;
    feedItem.Author = item.Authors[0].Name.ToString();
    // Handle the differences between RSS and Atom feeds.
    if (feed.SourceFormat == SyndicationFormat.Atom10)
    {
        feedItem.Content = item.Content.Text;
        feedItem.Link = new Uri("http://windowsteamblog.com" + item.Id);
    }
    else if (feed.SourceFormat == SyndicationFormat.Rss20)
    {
        feedItem.Content = item.Summary.Text;
        feedItem.Link = item.Links[0].Uri;
    }
    feedData.Items.Add(feedItem);
}
return feedData;
```

return ステートメントに到達しても、FeedData オブジェクトが実際に返されるわけではありません。await ステートメントの直後にメソッドが呼び出し元に戻ったときは、メソッドの最終的な結果を表すための **Task** が返される点に注意してください。次のコードで、最終的に結果を取得します。 return feedData; の行で、待機している **Task** にメソッドの結果である FeedData オブジェクト~~を渡します~~が渡されます(Task オブジェクトへの通知は、プログラマーは意識しなくてよい。)。

Task は、GetFeedsAsync メソッドのこの行で待機されていました。

コメント [Y4]: 待機と言っても、UI スレッドは動いている状態。

C#

```
this.Feeds.Add(await feed1);
```

Task が、待機していた FeedData の結果を取得すると、コードの実行が続行されて、FeedData が FeedDataSource.Feeds コレクションに追加されます。

アプリでのデータの使用

アプリでデータを使うには、App.xaml にリソースとしてデータ ソースのインスタンスを作ります。このインスタンスに feedDataSource という名前を付けます。

▶アプリにリソースを追加するには

1. **ソリューション エクスプローラー**で、App.xaml をダブルクリックします。XAML エディターでファイルが開きます。
2. **MergedDictionaries** コレクション内に <ResourceDictionary> </ResourceDictionary> タグを追加します。
3. 新しい **ResourceDictionary** にリソースの宣言、<local:FeedDataSource x:Key="feedDataSource"/> を追加します。

XAML (App.xaml)

```
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <!--
                Styles that define common aspects of the platform look and feel
                Required by Visual Studio project and item templates
             -->
            <ResourceDictionary Source="Common/StandardStyles.xaml"/>

            <!-- Add app resources to this resource dictionary -->
            <ResourceDictionary>
                <local:FeedDataSource x:Key="feedDataSource"/>
            </ResourceDictionary>

        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Application.Resources>
```

書式変更: 蛍光ペン

ページ テンプレートの分離コード ファイル(MainPage.xaml.cs)には、**OnNavigatedTo** メソッドのオーバーライドが既に含まれています。このメソッドにコードを記述し、アプリの FeedDataSource インスタンスを作成してフィードを取得します。まず、**async** キーワードをメソッドの宣言に追加します。これはメソッド内で **await** キーワードを使うからです。ページに移動したら(OnNavigatedTo メソッドが呼び出される)、FeedDataSource にフィードが既に含まれているかどうかを確認します(①)。含まれていない場合は、FeedDataSource.GetFeedsAsync メソッドを呼び出します(②)。次に、ページの **DataContext** ~~を~~に最初のフィードに設定します(③)。 次に示すのは、MainPage.xaml.cs に関連するコードです。
※ 黄色マーカーの部分を、コピー&ペーストします。

C# (MainPage.xaml.cs)

```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    FeedDataSource feedDataSource
            = (FeedDataSource)App.Current.Resources["feedDataSource"];
```

書式変更: 蛍光ペン

書式変更: 蛍光ペン

```
            if (feedDataSource != null)
            {
                if (feedDataSource.Feeds.Count == 0) //①
                {
                    await feedDataSource.GetFeedsAsync();//②
                }
                this.DataContext = (feedDataSource.Feeds).First(); //③
            }
        }
```

書式変更: 蛍光ペン

書式変更: 蛍光ペン

書式変更: 蛍光ペン

エラーなしでビルドされ、フィードが取得されるように、デバッグ モードでアプリをビルドし、実行できます。アプリをデバッグ モードで実行するには、**[デバッグ]**、**[デバッグ開始]** の順にクリックするか、F5 キーを押します。エラーがなければアプリは実行され、黒い画面だけが表示されます。Visual Studio に戻るには、Alt キーを押しながら Tab キーを押します。Visual Studio で、**[デバッグ]**、**[デバッグの停止]** の順にクリックしてアプリを閉じます。エラーがある場合は、エラーに関する情報が表示されます。Visual Studio でのアプリの実行について詳しくは、Visual Studio からの Metro スタイル アプリの実行に関するページをご覧ください。

XAML でのアプリのレイアウトの定義

注 次の 3 つのセクション、「XAML でのアプリのレイアウトの定義」、「コントロールとコンテンツの追加」、「データの表示」では、XAML でのユーザー インターフェイス作成の基本を見直します。これらの基本を身に付けるために、単一のブログ フィードの投稿を表示する単純な 1 ページのブログ リーダーを作ります。既に XAML の使用経験があり、XAML のレイアウト、コントロール、データ バインドを理解している場合は、これらのセクションの演習を完了せずに省略してかまいません。ただし、日付コンバーター用のコードの追加は省略しないでください (後ほど、アプリで使います)。「ページとナビゲーションの追加」では、Metro スタイル アプリ全体の作成に戻ります。

書式変更: 蛍光ペン

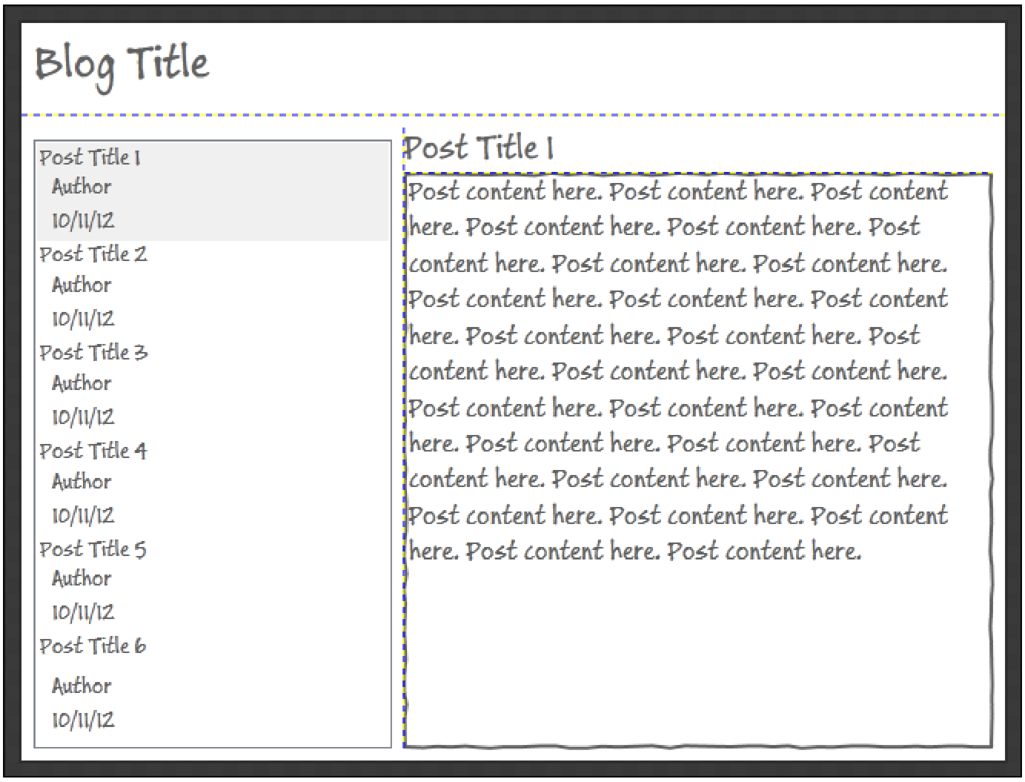
このセクションで説明する内容:

- XAML でレイアウトを定義するために使用できるパネル
- **Grid** で行と列を定義する方法
- **StackPanel** を使う方法

アプリのレイアウトで、アプリ内の各オブジェクトのサイズと位置を指定します。ビジュアル オブジェクトを配置するには、**Panel** コントロールなどのコンテナー オブジェクトにビジュアル オブジェクトを配置する必要があります。XAML レイアウト システムには、コントロールを配置できるコンテナーとしての役割を持つさまざまな **Panel** コントロール (**Grid**、**Canvas**、**StackPanel** など) が用意されています。

XAML レイアウト システムでは、絶対レイアウトと動的レイアウトの両方がサポートされます。絶対レイアウトでは、明示的な XY 座標を使ってコントロールを配置します (**Canvas** などを使います)。動的レイアウトでは、アプリのサイズが変更されたときにレイアウト コンテナーとコントロールのサイズと位置を自動的に調整できます (**StackPanel**、**Grid** などを使います)。現実にアプリのレイアウトを定義する方法としては、絶対レイアウトと動的レイアウトを組み合わせ、パネルを他のパネル内に埋め込むやり方が一般的です。

ブログ リーダー アプリの一般的なレイアウトでは、次の図に示すように、最上部にタイトル、左側に投稿の一覧、選んだ投稿の内容が右側に表示されます。



空白のアプリ テンプレートには、既定で UI のルート要素である空の **Grid** のみが含まれています。目標のレイアウトを指定するために、**Grid** を 2 つの行に分割します。上の行には、ブログのタイトルを格納します。2 番目の行では、別の **Grid** を埋め込んだ後、これを 2 つの列に分割します。さらにブログの内容を表示するためのレイアウト コンテナーをいくつか追加します。

※ 黄色マーカーの部分を、コピー&ペーストします。

XAML (MainPage.xaml)

```
    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <Grid.RowDefinitions>
            <RowDefinition Height="140" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <!-- Title -->

        <!-- Content -->
        <Grid Grid.Row="1">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="2*" MinWidth="320" />
                <ColumnDefinition Width="3*" />
            </Grid.ColumnDefinitions>
```

書式変更: 蛍光ペン

```
            <!-- Left column -->
            <ListView x:Name="ItemListView" />

            <!-- Right column -->
            <Grid Grid.Column="1" >
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto" />
                    <RowDefinition Height="*" />
                </Grid.RowDefinitions>
            </Grid>
        </Grid>
    </Grid>
```

それでは、この XAML の内容を詳しく見てみましょう。**Grid** に行を定義するために、**RowDefinition** オブジェクトを **Grid.RowDefinitions** コレクションに追加します。**RowDefinition** のプロパティを指定すると、行の体裁を指定できます。同様に、**ColumnDefinition** オブジェクトと **Grid.ColumnDefinitions** コレクションを使って列を追加します。

XAML での行定義は次のようになります。

XAML

```
<Grid.RowDefinitions>
  <RowDefinition Height="140"/>
  <RowDefinition Height="*"/>
</Grid.RowDefinitions>
```

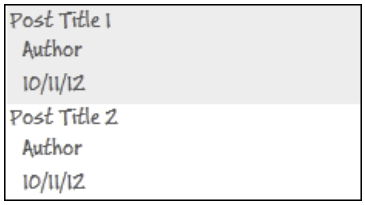
1 つ目の行定義 (行 0) の Height="140" プロパティ設定では、上の行の高さを 140 dip (デバイスに依存しないピクセル数) に固定しています。この高さは、行の内容やアプリのサイズに関係なく、変化しません。2 つ目の行定義 (行 1) の Height="*" 設定は、行 0 が割り当てられた後のすべての領域を下の行に割り当てることを指定しています。このような指定方法は、"スター サイズ指定" とも呼ばれます。スター サイズ指定は、2 つ目の **Grid** の列定義でも使います。Width="2*" と Width="3*" の幅設定は、**Grid** を 5 等分したうえで、そのうちの 2 つの部分を 1 つ目の列に使い、3 つの部分を 2 つ目の列に使うことを指定しています。

要素を **Grid** 内に配置するには、要素の **Grid.Row** 添付プロパティと **Grid.Column** 添付プロパティを設定します。行と列の番号は 0 から始まります。これらのプロパティの既定値は 0 なので、何も設定しない場合、要素は 1 つ目の行の 1 つ目の列に位置付けられます。

<Grid Grid.Row="1"> 要素は、**Grid** をルートの **Grid** の最下行に埋め込むことを表します。この **Grid** は、2 つの列に分割されます。

<ListView x:Name="ItemListView"> 要素は、**ListView** を下部の **Grid** の左側の列に追加します。<Grid Grid.Column="1"> 要素は、別の **Grid** を下部の **Grid** の右側の列に追加します。この **Grid** を 2 つの行に分けます。Height="Auto" 設定は、上の行の高さを、その内容が収まるように調整することを表します。下の行は、残りの領域を使います。

レイアウト パネルを必要とする UI の最後の部分は、ブログへの投稿の一覧です。この一覧には、次に示すように、タイトル、作成者、日付を含めます。



通常、ページ上の UI の小さなサブセクションに連続する要素を自動的に配置するには、**StackPanel** を使います。**StackPanel** は、子要素が単一の行に水平方向または垂直方向に配置される単純なレイアウト パネルです。子要素を並べる向きを指定するには、**StackPanel.Orientation** プロパティを使います。**Orientation** プロパティの既定値は **Orientation.Vertical** です。ここでは、**StackPanel** を使って、ブログの投稿の一覧の項目を並べます。使用例については、「データ テンプレートを使ったデータの書式設定」をご覧ください。**StackPanel** の XAML は次のようになります。

XAML（このコードは、まだ記述しない! 後に詳しい説明がある。）

```
<StackPanel>
    <TextBlock Text="{Binding Path=Title}" FontSize="24" Margin="5,0,0,0"
TextWrapping="Wrap" />
    <TextBlock Text="{Binding Path=Author}" FontSize="16" Margin="15,0,0,0"/>
    <TextBlock Text="{Binding Path=PubDate}" FontSize="16" Margin="15,0,0,0"/>
</StackPanel>
```

コントロールとコンテンツの追加

このセクションの内容:

- コントロールをアプリに追加する方法

レイアウト パネルは重要ですが、その中にコンテンツをどのように配置するかが肝心です。 ここでは、ボタン、リスト、テキスト、グラフィックス、イメージなどのコントロールを追加して、アプリの UI を作ります。どのような要素を使うかは、アプリで何をするかによって異なります。Metro スタイル アプリで使用できるコントロールの全一覧については、コントロールの一覧を参照してください。

ブログ リーダー UI では、図に示すように、1 行テキスト (ブログのタイトルと投稿のタイトル)、複数行テキスト (投稿の内容)、ブログの投稿の一覧を表示する必要があります。そこで、タイトルを表示する **TextBlock** コントロールと、ブログの投稿の一覧を表示する **ListView** コントロールを追加します。投稿の内容を表示する方法として、複数行 **TextBlock** または **RichTextBlock** を使う方法が最初に考えられます。しかし、よく調べると、投稿の内容を含む文字列はプレーンテキストではなく HTML の文字列であることがわかります。そのような文字列を **TextBlock** に入れると、大量の HTML タグが表示される結果になります。このような事態を避けるためには、**WebView** コントロールを使って HTML を表示します。

コントロールを追加した後の UI の XAML は次のようになります。
※ 黄色マーカーが、変更した部分。

XAML (MainPage.xaml)

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

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <Grid.RowDefinitions>
            <RowDefinition Height="140" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <!-- Title -->
        <TextBlock x:Name="TitleText" Text="{Binding Title}"
```

書式変更: フォント : 太字
書式変更: フォント : 太字, 蛍光ペン

```
                VerticalAlignment="Center" FontSize="48" Margin="56,0,0,0"/>

        <!-- Content -->
        <Grid Grid.Row="1">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="2*" MinWidth="320" />
                <ColumnDefinition Width="3*" />
            </Grid.ColumnDefinitions>

            <!-- Left column -->
            <!-- The default value of Grid.Column is 0, so we do not need to set it
                 to make the ListView show up in the first column. -->
            <ListView x:Name="ItemListView"
                      ItemsSource="{Binding Items}"
                      Margin="60,0,0,10">
            </ListView>

            <!-- Right column -->
            <!-- We use a Grid here instead of a StackPanel so that the WebView sizes
correctly. -->
            <Grid DataContext="{Binding ElementName=ItemListView, Path=SelectedItem}"
                  Grid.Column="1" Margin="25,0,0,0">
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto" />
                    <RowDefinition Height="*" />
                </Grid.RowDefinitions>
                <TextBlock x:Name="PostTitleText" Text="{Binding Title}"
                           FontSize="24"/>
                <WebView x:Name="ContentView" Grid.Row="1" Margin="0,5,20,20"/>
            </Grid>
        </Grid>
    </Grid>
</Page>
```

書式変更: フォント : 太字

書式変更: フォント : 太字, 蛍光ペン

書式変更: フォント : 太字, 蛍光ペン

書式変更: フォント : 太字

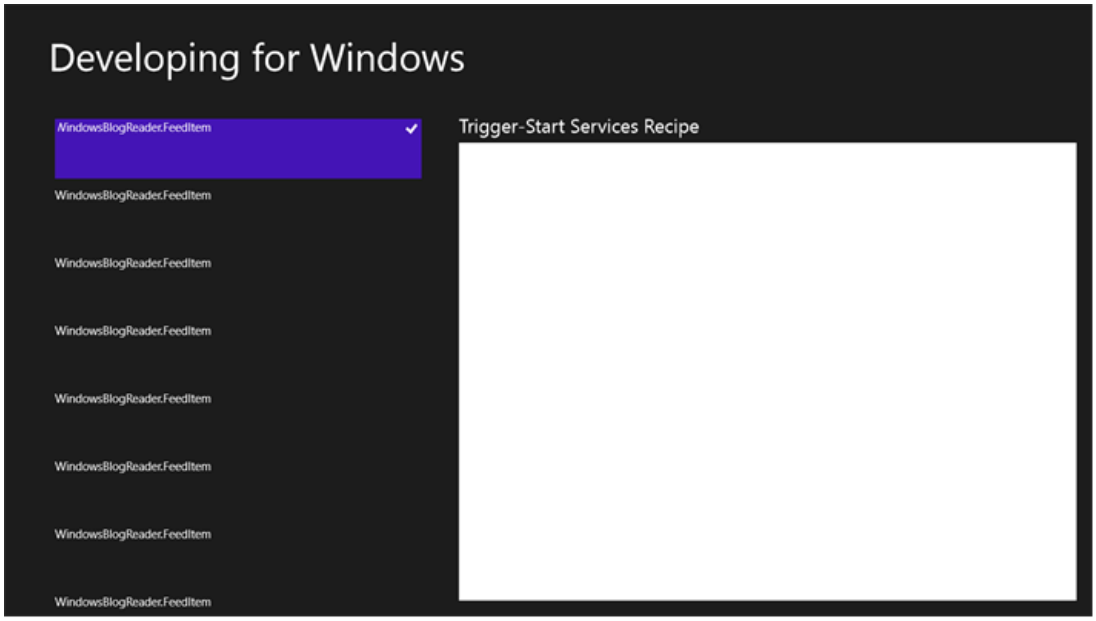
書式変更: フォント : 太字, 蛍光ペン

書式変更: フォント : 太字

書式変更: フォント : 太字, 蛍光ペン

書式変更: フォント : 太字

ここで F5 キーを押してデバッグ モードでアプリをビルドして実行し、アプリの UI を確認することができます。内容はまだあまりありませんが、エラーがなければ次のような画面が表示されます。Visual Studio に戻るには、Alt キーを押しながら Tab キーを押します。Visual Studio で、**[デバッグ]**、**[デバッグの停止]** の順にクリックしてアプリを閉じます。



※ F5 で上手く実行できても、しばらくは何も画面に出てきません。非同期で RSS を取得しているからです。

※ 画面が出たら、左のリストを選択してみてください。右側のテキストも変わります。

データの表示

このセクションの内容:

- データを UI にバインドする方法
- テンプレートを使ってデータを書式設定する方法

UI へのデータのバインド

最初に紹介した Hello World アプリでは、UI 内のテキストを更新するために次のボタン クリック イベント ハンドラー内で **TextBlock** の **Text** プロパティを設定しました。

C# (冒頭の "Hello World")

```
private void HelloButton_Click(object sender, RoutedEventArgs e)
{
    DisplayText.Text = "Hello World";
}
```

このように **Text** プロパティをコードで設定する方法がときにはうまく機能することもあります。しかし、通常は、データを表示するために、データ バインドを使ってデータ ソースを UI に接続します。バインドを確立した場合、データ ソースが変更されたときに、データ ソースにバインドされている UI 要素でその変更を自動的に反映できます。同様に、ユーザーによって UI 要素に加えられた変更をデータ ソースに反映することもできます。たとえば、ユーザーが **TextBox** の値を編集した場合、その変更を反映するためにバインド エンジンによって基のデータ ソースが自動的に更新されます。

このブログ リーダー アプリでは、タイトル **TextBlock** の **Text** プロパティをソース オブジェクトの **Title** プロパティにバインドして、ブログ タイトルを表示します。

XAML (MainPage.xaml、記述済み)

```
<TextBlock x:Name="TitleText" Text="{Binding Title}"
           VerticalAlignment="Center" FontSize="48" Margin="56,0,0,0" />
```

選ばれたブログの投稿のタイトルも同じ方法で表示します。

XAML (MainPage.xaml、記述済み)

```
<TextBlock x:Name="PostTitleText" Text="{Binding Title}" FontSize="24"/>
```

しかし少し待ってください。両方の **TextBlock** がまったく同じバインドを持つ場合、異なるタイトルを表示するにはどうしたらよいでしょうか(実際、異なる文字列が表示されています!)。その答えは、それぞれの **TextBlock** がバインドされている **DataContext** にあります。**DataContext** プロパティでは、すべての子要素を含む UI 要素全体の既定のバインドを設定できます。**DataContext** プロパティをページ全体に対して設定することも、ページ上の個々の要素に対して設定することもできます。それぞれの XAML レベルの **DataContext** 設定は、上位レベルの設定より優先されます。さらに、個々のバインドに対して有効な任意の **DataContext** 設定は、**Source** プロパティを設定することで上書きできます。

ブログ リーダー アプリでは、分離コード内で **DataContext** をページ全体に対して設定します。データ フィードを取得した後、次のコードを使って **DataContext** を設定したことを思い出してください。

C#

```
        protected override async void OnNavigatedTo(NavigationEventArgs e)
        {
            ...
            this.DataContext = (feedDataSource.Feeds).First();
        }
```

1 つ目の **TextBlock** のコンテキストは FeedData オブジェクトであるため、FeedData.Title プロパティが表示されます。

選ばれたブログの投稿のタイトルは、どのようにして 2 つ目の **TextBlock** で表示されるのでしょうか。次に示すように、2 つ目の **TextBlock** は **Grid** 内にあります。

XAML

```
<Grid DataContext="{Binding ElementName=ItemListView, Path=SelectedItem}"
      Grid.Column="1" Margin="25,0,0,0">
      ...
      <TextBlock x:Name="PostTitleText" Text="{Binding Title}" FontSize="24"/>
      ...
</Grid>
```

Grid の **DataContext** は **ListView** の **SelectedItem** プロパティにバインドされています。ここでも、陰でバインド エンジンが働いています。**ListView** 内の選択項目が変更されると、**StackPanel** の **DataContext** が選ばれた投稿に合わせて自動的に更新されます。**Grid** の

DataContext はページの **DataContext** より優先されるため、2 つ目の **TextBlock** には、選ばれたブログの投稿の FeedItem.Title プロパティが表示されます。

それぞれのバインドには、データ更新の方法とタイミングを指定する **Mode** プロパティがあります。

バインド モード	説明
OneTime	バインドが最初に作られたときにのみターゲットに値が設定され、それ以降は値が更新されません。
OneWay	ソースが変更された場合にターゲットが更新されます。
TwoWay	ターゲットとソースのどちらかが変更された場合にターゲットとソースのどちらも更新されます。

OneWay バインドまたは **TwoWay** バインドを使う場合、ソース オブジェクトに対する変更をバインドに通知するには、**INotifyPropertyChanged** インターフェイスを実装する必要があります。データ バインドについて詳しくは、コントロールへのデータ バインドに関するクイック スタート トピックをご覧ください。

データ テンプレートを使ったデータの書式設定

一覧ビューに目的のデータを表示する操作は、バインドを設定するよりもやや複雑です。**ListView** は FeedData オブジェクトの **Items** プロパティにバインドされているので、目的のデータはそこにあります。しかし、このようなアプリを実行した場合、**ListView** からは表示内容を認識できないため、単にバインドされているオブジェクトの **ToString** を呼び出します。その結果、前に見たような "**WindowsBlogReader.FeedItem**" 文字列の一覧が表示されます。これは、意図している動作ではありません。

これを修正するには、**ListView** の **DisplayMemberPath** プロパティを設定します。すると、一覧ビューは、バインドされたオブジェクト自体ではなく、その指定されたプロパティの **ToString** を呼び出します。DisplayMemberPath=Title を設定した場合、**ListView** にはブログの投稿のタイトルの一覧が表示されます。

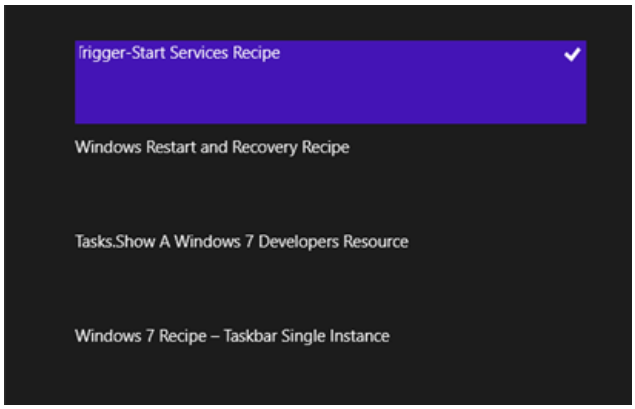
XAML

```
<ListView x:Name="ItemListView"
                  ItemsSource="{Binding Items}"
                  DisplayMemberPath="Title"
                  Margin="60,0,0,10">
```

書式変更: 蛍光ペン

```
</ListView>
```

ここで F5 キーを押し、アプリをビルドして実行します。一覧の項目は次のようになります。これで、目的の動作に近づきました。



しかし、本来の目的は、各投稿のタイトル、作成者、公開日を一覧に表示することです。そのためには、どのようにデータを表示するかを明確に **ListView** に指示するテンプレートを定義します。データ項目のコレクションを表示するためのコントロールは、**ItemsControl** クラスから派生します。これらのコントロールの **ItemTemplate** プロパティに、**DataTemplate** を割り当てることができます。**DataTemplate** は、データの表示方法を定義します。

重要 **DisplayMemberPath** と **ItemTemplate** を同時に使うことはできません。**ItemTemplate** を **ListView** に追加する場合は、必ず **DisplayMemberPath** の設定を削除してください。(両方あると、実行時エラーになる。)

次に示す **ListView** の XAML では、~~**DataTemplate** の代わりに **DisplayMemberPath**~~ DisplayMemberPath を削除して、代わりに ItemTemplate がインラインで定義されています。

XAML

```
<ListView x:Name="ItemListView"
          ItemsSource="{Binding Items}"
          Margin="60,0,0,10"><!-- DisplayMemberPath="Title" は削除する -->
    <ListView.ItemTemplate>
        <DataTemplate>
            <StackPanel>
                <TextBlock Text="{Binding Title}"
                           FontSize="24" Margin="5,0,0,0" TextWrapping="Wrap" />
                <TextBlock Text="{Binding Author}"
                           FontSize="16" Margin="15,0,0,0"/>
                <TextBlock Text="{Binding PubDate}"
```

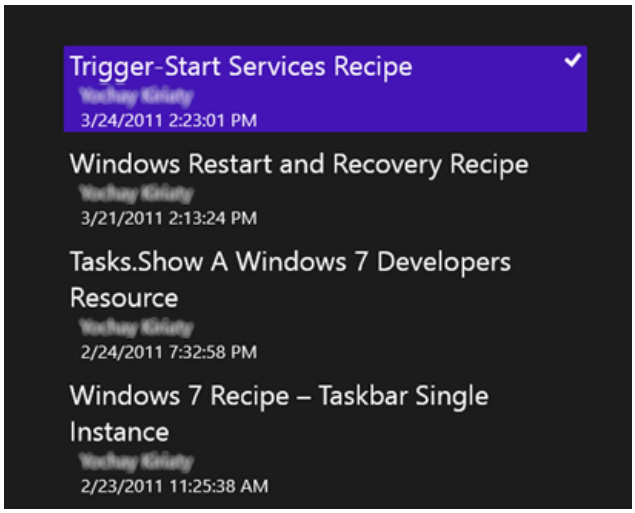
書式変更: 蛍光ペン

書式変更: フォント : 太字, 蛍光ペン

```
                    FontSize="16" Margin="15,0,0,0"/>
            </StackPanel>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

DataTemplate 内での UI の定義は、通常の UI の定義と似ています。ここでは、StackPanel を使って、3 つの TextBlock を重ね合わせています。次に、**TextBlock** の Text プロパティを **Title** プロパティ、**Author** プロパティ、**PubDate** プロパティにバインドします。これらのバインドの既定の DataContext は、ListView に表示されるオブジェクトで、それは FeedItem です。

F5 キーを押して、アプリをビルドし、実行します。テンプレートを使って外観を定義した場合、一覧は次のように表示されます。



値コンバーターを使ったデータの型指定

ItemListView の DataTemplate では、**PubDate** プロパティ (DateTime 型) を TextBlock.Text プロパティにバインドします。バインド エンジンによって、**PubDate** の型が **DateTime** から文字列へと自動的に変換されます。この自動変換では日付と時刻の両方が表示されますが、ここでは日付のみを表示したいとします。そこで、**DateTime** を文字列に変換する値コンバーターを独自に作り、その中で自由に文字列を書式設定できるようにします。

値コンバーターを作成するには、IValueConverter インターフェイスを実装するクラスを作成した後で、Convert メソッドと ConvertBack メソッドを実装します。コンバーターでは、データ

の型の変更、カルチャ情報に基づいたデータの変換、その他の表示の変更を実行できます。ここでは、渡された日付値を変換して、年、月、日のみが表示される形式にする日付コンバーターを作成します。

注 プロジェクトに新しいクラスを追加するには、**[プロジェクト]** の **[クラスの追加]** をクリックします。クラスの名前を **DateConverter** (.cs または .vb) にします。

Convert メソッドと **ConvertBack** メソッドではパラメーターを渡すことができるため、コンバーターの同じインスタンスで異なるオプションを使うことができます。次の例では、入力パラメーターに基づいて異なる日付書式を生成する書式設定コンバーターを作成しています。**Binding** クラスの **ConverterParameter** を使うと、パラメーターを **Convert** メソッドと **ConvertBack** メソッドに引数として渡すことができます。この方法を示すために、後で一覧ビューに変更を加えます。

※ DateConverter.cs ファイルを追加し、次のコードをコピー&ペーストします。

C# (新しく追加した DateConverter.cs)

```
using System;
using Windows.Globalization.DateTimeFormatting;

namespace WindowsBlogReader
{
    public class DateConverter : Windows.UI.Xaml.Data.IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, string
culture)
        {
            if (value == null)
                throw new ArgumentNullException("value", "Value cannot be null.");

            if (!typeof(DateTime).Equals(value.GetType()))
                throw new ArgumentException("Value must be of type DateTime.",
"value");

            DateTime dt = (DateTime)value;

            if (parameter == null)
            {
                // Date "7/27/2011 9:30:59 AM" returns "7/27/2011"
                return DateTimeFormatter.ShortDate.Format(dt);
            }
            else if ((string)parameter == "day")
            {
                // Date "7/27/2011 9:30:59 AM" returns "27"
```

```
                DateTimeFormatter dateFormatter = new
DateTimeFormatter("{day.integer(2)}");
                return dateFormatter.Format(dt);
            }
            else if ((string)parameter == "month")
            {
                // Date "7/27/2011 9:30:59 AM" returns "JUL"
                DateTimeFormatter dateFormatter = new
DateTimeFormatter("{month.abbreviated(3)}");
                return dateFormatter.Format(dt).ToUpper();
            }
            else if ((string)parameter == "year")
            {
                // Date "7/27/2011 9:30:59 AM" returns "2011"
                DateTimeFormatter dateFormatter = new DateTimeFormatter("{year.full}");
                return dateFormatter.Format(dt);
            }
            else
            {
                // Requested format is unknown. Return in the original format.
                return dt.ToString();
            }
        }

        public object ConvertBack(object value, Type targetType, object parameter,
string culture)
        {
            string strValue = value as string;
            DateTime resultDateTime;
            if (DateTime.TryParse(strValue, out resultDateTime))
            {
                return resultDateTime;
            }
            return Windows.UI.Xaml.DependencyProperty.UnsetValue;
        }
    }
}
```

DateConverter クラスが追加されているため、F7 キーを押すと、エラーなしでソリューションがビルドされます。

DateConverter クラスを使うには、あらかじめ XAML でこのクラスのインスタンスを宣言しておく必要があります。アプリのリソースとして dateConverter キーを使って、App.xaml でインスタンスを宣言します。ここに宣言しておくと、使うアプリのどのページでもインスタンスを使うことができます。

XAML (App.xaml)

```
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            ...
            <ResourceDictionary>
                <local:FeedDataSource x:Key="feedDataSource"/>

                <!-- Add the DateConverter here. -->
                <local:DateConverter x:Key="dateConverter" />

            </ResourceDictionary>

        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Application.Resources>
```

書式変更: フォント : 太字, 蛍光ペン
書式変更: フォント : 太字

これで、バインドに DateConverter を使用できるようになりました。次に、ItemListView の DataTemplate に含まれている **PubDate** の更新されたバインドを示します。

XAML (MainPage.xaml)

```
<TextBlock Text="{Binding Path=PubDate, Converter={StaticResource dateConverter}}"
           FontSize="16" Margin="15,0,0,0"/>
```

書式変更: フォント : 太字, 蛍光ペン

この XAML の場合、バインド エンジンでカスタム DateConverter を使って、DateTime を文字列に変換します。返される文字列は、意図したとおり、年、月、日のみが含まれる形式になっています。

F5 キーを押して、アプリをビルドし、実行します。一覧ビューに時間が表示されなくなります。

WebView での HTML の表示

ページにブログの投稿を表示する最後の手順は、投稿データを取得して WebView コントロールに表示することです。このコントロール、つまり ContentView は、既に UI に追加しています。

XAML (MainPage.xaml の下の方)

```
<WebView x:Name="ContentView" Grid.Row="1" Margin="0,5,20,20"/>
```

WebView コントロールを使うと、アプリ内で HTML データをホストできます。しかし、その Source プロパティを見ると、表示する Web ページの Uri を受け取ることがわかります。ここで扱う HTML データは、単純な HTML 文字列です。**Source** プロパティにバインドできる **Uri** は含まれていません。幸運なことに、NavigateToString メソッドがあります。このメソッドに HTML の文字列を渡すことができます。

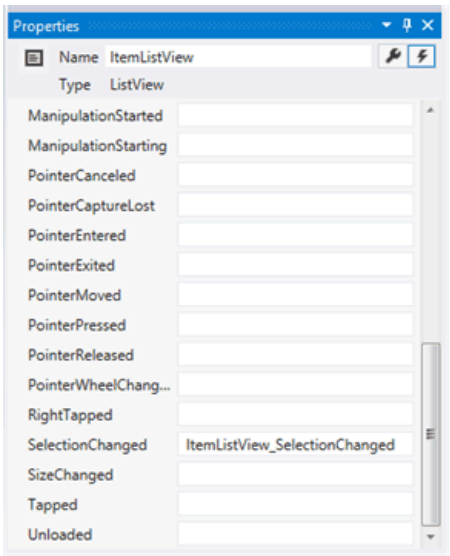
そのために、ListView の SelectionChanged イベントを処理します。

▶イベント ハンドラーを追加するには

1. XAML ビューまたはデザイン ビューで、イベントを処理する必要があるコントロールを選びます。この例では、MainPage.xaml の `ItemListView` をクリックします。
2. **プロパティ ウィンドウ**の [イベント] ボタン () をクリックします。

   ヒント プロパティ ウィンドウが表示されない場合は、Alt キーを押しながら Enter キーを押して、プロパティ ウィンドウを開きます。

3. イベントの一覧で SelectionChanged イベントを探します。このイベントのボックスに「"ItemListView_SelectionChanged"」と入力します。



4. Enter キーを押します。イベント ハンドラーが作られ、コード エディターで開きます。イベント発生時に実行するコードを追加します。

SelectionChanged イベントを追加した ListView の XAML を次に示します。

XAML (MainPage.xaml)

```
<ListView x:Name="ItemListView"
                  ItemsSource="{Binding Items}"
                  Margin="60,0,0,10"
                  SelectionChanged="ItemListView_SelectionChanged">
    ...
```

分離コード ページで作られたイベント ハンドラーにコードを追加します。イベント ハンドラーでは、選んだ項目を FeedItem にキャストし(①)、**Content** プロパティから HTML 文字列を取得します。その文字列を、**NavigateToString** メソッドに渡します(②)。項目が選ばれていない場合は(③)、空の文字列を渡して **WebView** をクリアします(④)。

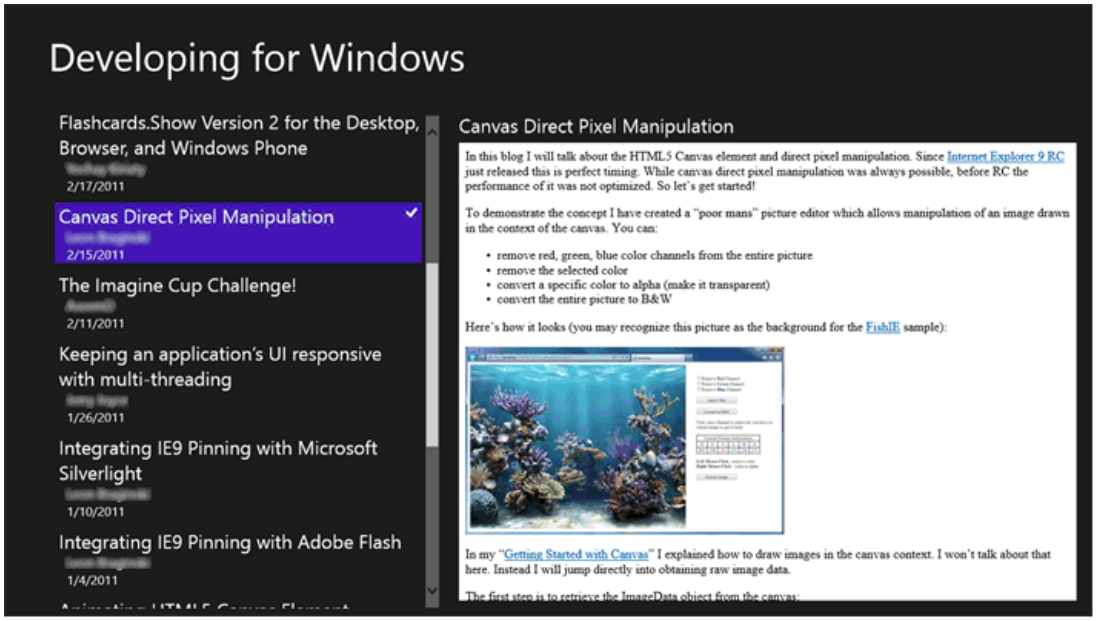
※ メソッドの中身をコピー&ペーストします。

C#

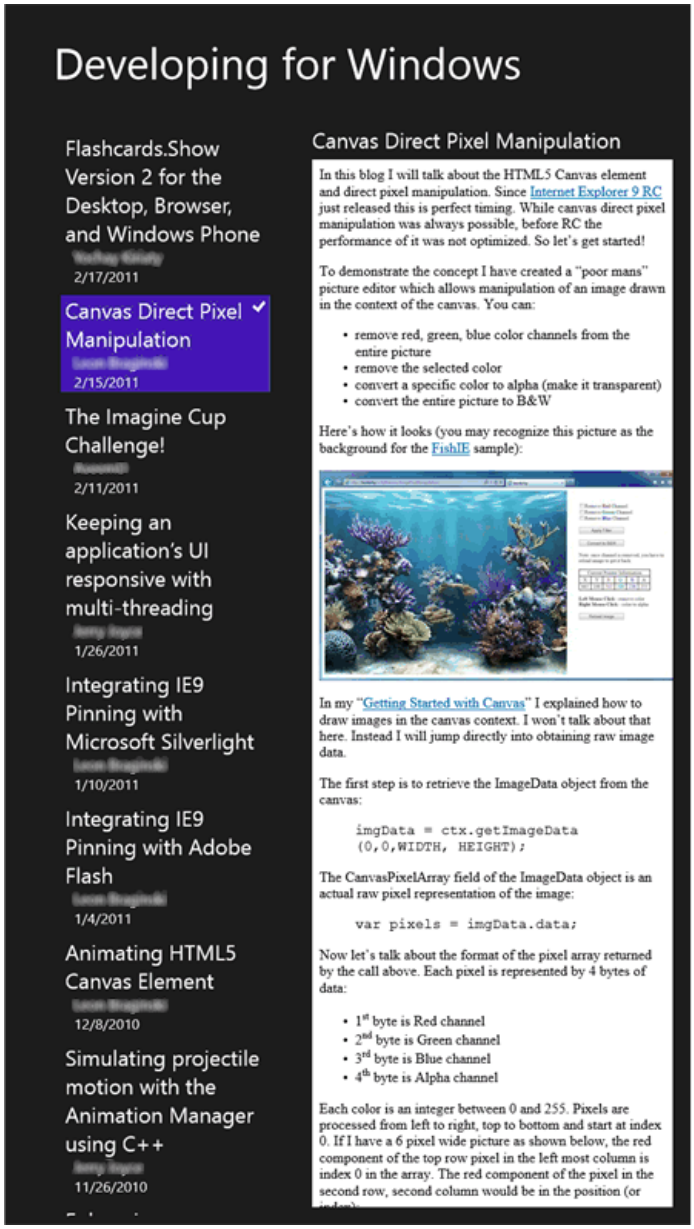
```
private void ItemListView_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
            // If there's a selected item (in AddedItems)
            // show it in the WebView.
            if (e.AddedItems.Count > 0)
            {
                FeedItem feedItem = e.AddedItems[0] as FeedItem;  //①
                if (feedItem != null)
                {
                    // Navigate the WebView to the blog post content HTML string.
                    ContentView.NavigateToString(feedItem.Content);  //②
                }
            }
            else  //③
            {
                // If the item was de-selected, clear the WebView.
                ContentView.NavigateToString("");  //④
            }
}
```

書式変更: フォント : 太字
書式変更: フォント : 太字, 蛍光ペン
書式変更: フォント : 太字, 蛍光ペン
書式変更: フォント : 太字, 蛍光ペン
書式変更: フォント : 太字, 蛍光ペン
書式変更: フォント : 太字, 蛍光ペン
書式変更: フォント : 太字

このアプリを実行すると、次のように表示されます。



この基本ページは、ほとんどの状況で正常に動作します。しかし、Metro スタイル アプリは、あらゆる状況で正常に動作することが求められます。さまざまなデバイスのさまざまな解像度、向き、ビューに対応する必要があります。たとえば、この図は、ページを縦方向に表示したときのようすを示しています。



この基本ページはこのように表示したときにうまく動作しないことがわかります。アプリの外観を良くすると共に、Windows チーム ブログの雰囲気に合うようにしたいとします。これらの目標を満たすために、最初に Microsoft Visual Studio に付属するページ テンプレートについて説明します。次に、このページを作成する中で学習した内容に基づいてテンプレートを変更して、アプリを完成させます。

ページとナビゲーションの追加

このブログ リーダーですべての Windows チーム ブログに対応するためには、アプリにさらにページを追加し、そのページ間でのナビゲーションを処理する必要があります。まず、すべての Windows チーム ブログを一覧表示するページが必要です。ユーザーがページでブログを選ぶと、そのブログの投稿一覧を読み込みます。これまでに作成した分割ページ リーダーでもこの目的に対応できますが、ここではさらに改善を加えることにします。最後に、一覧ビューを開かずに個別のブログ投稿を読むことができるように、詳細ビューを追加します。

ページ テンプレート

それぞれのページを空のテンプレートから作成する必要はありません。Visual Studio Express 2012 RC for Windows 8 には、さまざまな状況に役立つ多様なページ テンプレートが付属しています。利用可能なページ テンプレートは次のとおりです。

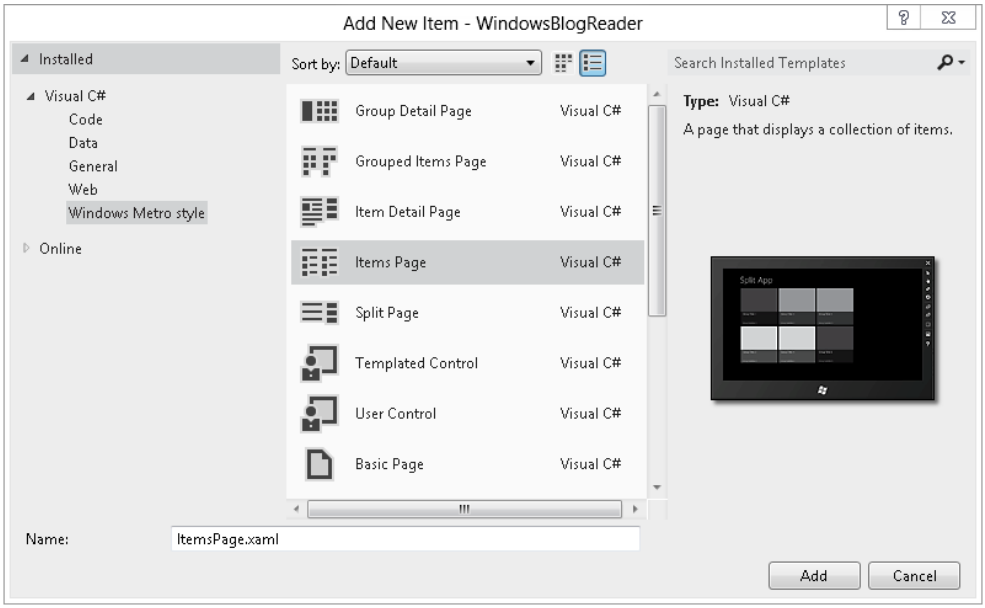
ページの種類	説明
グループ~~の~~詳細ページ	単一のグループの詳細と、グループ内の各項目のプレビューを表示します。
グループ化された項目ページ	グループ化されたコレクションを表示します。
~~項目の~~アイテム詳細ページ	1 つの項目を詳しく表示し、隣接する項目へのナビゲーションを有効にします。
項目ページ	項目のコレクションを表示します。
分割ページ	項目の一覧と、選ばれた項目の詳細を表示します。
基本ページ	さまざまな向きとビューに対応でき、タイトルと戻るボタンを備えた空白のページです。
空白のページ	Metro スタイル アプリ用の空白のページです。

アプリにページを追加するには

1. **[プロジェクト]** の **[新しい項目の追加]** をクリックします。**[新しい項目の追加]** ダイアログ ボックスが開きます。
2. **[インストール済み]** ウィンドウで、**[Visual C#]** または **[Visual Basic]** を展開します。
3. テンプレートの種類として **[Windows Metro ~~スタイル~~ Style]** を選びます。
4. 中央のウィンドウで、プロジェクトに追加するページの種類を選びます。
5. ページの名前を入力します。

6. **[追加]** をクリックします。ページの XAML と分離コード ファイルがプロジェクトに追加されます。

   [新しい項目の追加] ダイアログを次に示します。



7. "~~新しいアプリ~~ Blank App" テンプレートに**空白のページ**以外の新しいページを初めて追加すると、プロジェクトに足りないファイルを追加する必要があることを通知するメッセージ ボックスが表示されます。**[はい]** をクリックしてこれらのファイルを追加します。さまざまなユーティリティ クラスのファイルが、プロジェクトの **Common** フォルダーに追加されます。
8. F7 キーを押してアプリをビルドします。依存するヘルパー クラスをビルドするまで、デザイナーには新しいページのエラーが表示されます。

このブログ リーダーでは、Windows チーム ブログの一覧を表示するために**項目ページ**を追加します。ページの名前は ItemsPage とします。 各ブログの投稿を表示するための **Split Page** (分割ページ) を追加します。ページの名前は SplitPage とします。**Split Page** (分割ページ) テンプレートは、単純なブログ リーダー アプリ用に作成したページと似ていますが、はるかに洗練されています。次に、**基本ページ** テンプレートを詳細ページに使います。ページの名前は DetailPage とします。このページでは、戻るボタン、ページ タイトル、投稿の内容を表示する WebView コントロールしか使っていません。ただし、分割ページで行ったように HTML の文字列から **WebView** に投稿の内容を読み込む代わりに、投稿の URL に移動して実際の Web ページを表示します。完成したアプリのページは次のようになります。

書式変更: 蛍光ペン
書式変更: 蛍光ペン
書式変更: 蛍光ペン
書式変更: 蛍光ペン
書式変更: 蛍光ペン
書式変更: 蛍光ペン

ページ テンプレートをプロジェクトに追加し、XAML と分離コードを見ると、これらのページ テンプレートにさまざまな操作が組み込まれていることがわかります。実際、わかりにくい部分もあるので、ページ テンプレートの内容を詳しく見てみましょう。Metro スタイル アプリのページ テンプレートはすべて同じ形式です。

ページ テンプレートの XAML には、次の 3 つの主要セクションがあります。

リソース	リソース セクションには、ページのスタイルとデータ テンプレートが定義されています。これについては、「スタイルを使った外観の統一」で詳しく説明します。
アプリのコンテンツ	アプリの UI を構成するコントロールとコンテンツは、ルート レイアウト パネル内で定義します。
Visual State Manager	アプリをさまざまなレイアウトや向きに対応させるアニメーションと切り替えは、Visual State Manager (VSM) で定義されています。これについては、「さまざまなレイアウトへの対応」で詳しく説明します。

テンプレート ページはすべて LayoutAwarePage クラスから派生しており、最初に紹介した BlankPage よりもずっと多くの既定の操作があります。LayoutAwarePage は、Metro スタイル アプリ開発の重要な機能を実現する **Page** を実装したものです。

- アプリケーション ビュー状態を表示状態にマッピングすることで、ページをさまざまな解像度、向き、ビューに対応させることができます。
- GoBack イベント ハンドラーと GoHome イベント ハンドラーは基本的なナビゲーションをサポートします。
- 既定のビュー モデルは、単純でバインド可能なデータ ソースを提供しています。

また、ページ テンプレートでは、Metro スタイル アプリの設計ガイドラインに準拠した StandardStyles.xaml に記述されているスタイルとテンプレートを使います。冒頭部でこれらのスタイルのいくつかを使い、そのコピーを変更してアプリの外観をカスタマイズします。

ページ間のナビゲーション

XAML UI フレームワークには、**Frame** と **Page** を使って Web ブラウザーでのナビゲーションに似た操作を実現する組み込みのナビゲーション モデルが用意されています。**Frame** コントロールは **Page** をホストし、表示したページを前後に移動するためのナビゲーション履歴機能を持ちます。移動時は、ページ間でデータを渡すことができます。

Microsoft Visual Studio プロジェクト テンプレートでは、rootFrame という名前の **Frame** がアプリ ウィンドウのコンテンツとして設定されます。ここで、App.xaml.cs に関連するコードを見てみましょう。

C# (自動生成された App.xaml.cs)

```
protected override void OnLaunched(LaunchActivatedEventArgs args)
{
    // App state management code
    // ...

    // Create a Frame to act as navigation context and navigate to the first page
    var rootFrame = new Frame();
    if (!rootFrame.Navigate(typeof(MainPage)))
    {
        throw new Exception("Failed to create initial page");
    }

    // Place the frame in the current Window and ensure that it is active
    Window.Current.Content = rootFrame;
    Window.Current.Activate();
}
```

このコードでは、**Frame** を作成し、それをウィンドウのコンテンツとして設定して MainPage に移動します。完成したアプリの最初のページは ItemsPage になるので、次のように **Navigate** メソッドへの呼び出しを変更して ItemsPage ~~に~~を渡します。

C# (App.xaml.cs を修正する)

```
            if (!rootFrame.Navigate(typeof(ItemsPage)))
            {
                throw new Exception("Failed to create initial page");
            }
```

書式変更: フォント : 太字, 蛍光ペン

ItemsPage が読み込まれると、「アプリでのデータの使用」で MainPage を使って行ったように、データ ソースのインスタンスを作成して表示するフィード データを取得する必要があります。MainPage には、ページ テンプレートに含まれている **OnNavigatedTo** メソッドのオーバーライドをが配置しますされています。LayoutAwarePage から派生する新しいページでは、データは代わりに LoadState メソッドのオーバーライドで読み込まれます。ここで追加するコードは MainPage に追加したコードによく似ていますが、ページの **DataContext** を設定する代わりに、LayoutAwarePage に用意されている DefaultViewModel を使います。メソッド シグネチャに **async** キーワードを追加し、フィードがまだ取得されていない場合は FeedDataSource.GetFeedsAsync メソッドを呼び出します。次に示すのは、ItemsPage.xaml.cs に関連するコードです。

書式変更: 蛍光ペン

書式変更: 蛍光ペン

C# (ItemsPage.xaml.cs の LoadState()メソッドの中に記述する)

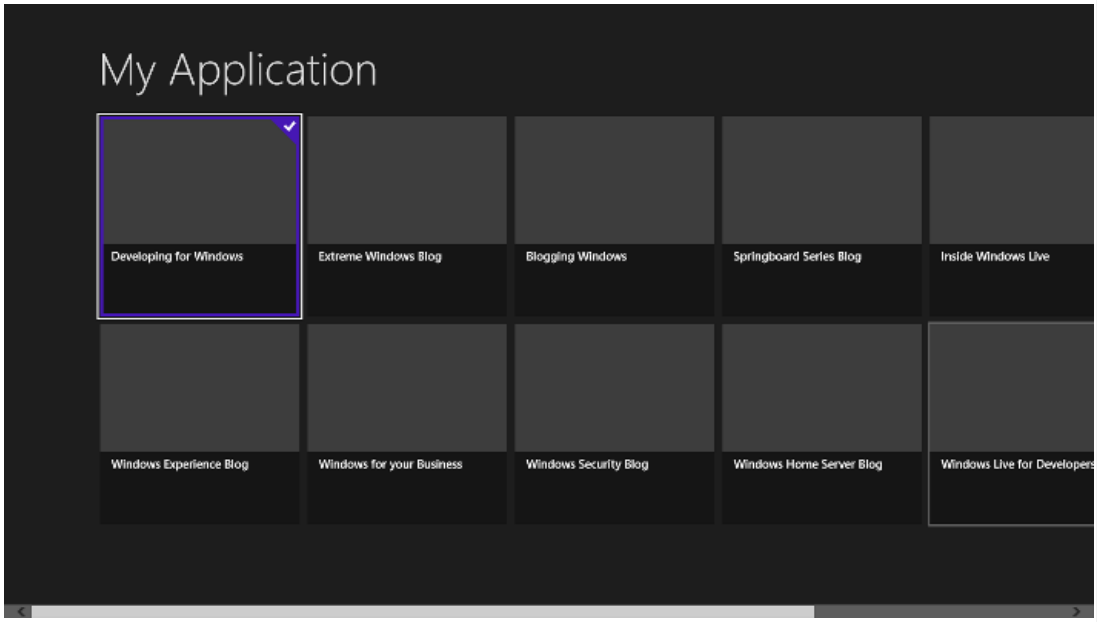
```
            FeedDataSource feedDataSource =
(FeedDataSource)App.Current.Resources["feedDataSource"];
            if (feedDataSource != null)
            {
                if (feedDataSource.Feeds.Count == 0)
                {
                    await feedDataSource.GetFeedsAsync();
                }
                this.DefaultViewModel["Items"] = feedDataSource.Feeds;
            }
```

書式変更: フォント : 太字

書式変更: フォント : 太字, 蛍光ペン

書式変更: フォント : 太字

F5 キーを押して、アプリをビルドし、実行します。前に作った MainPage の代わりに、ItemsPage が読み込まれます。このページにはすべてのブログのグリッドが表示され、既定で最初の項目が選択されます。表示は次のようになりますが、画面解像度によっては配置が変わる場合もあります。



ユーザーがコレクションからブログを選んだときに、項目ページから分割ページに移動するようにします。このナビゲーションを実現するには、選択ではなく、ボタンのようにクリックに応答する **GridView** 項目が必要です。**GridView** 項目をクリック可能にするために、次のように **SelectionMode** プロパティと **IsItemClickEnabled** プロパティを設定します。次に、**GridView** の **ItemClick** イベントのハンドラーを追加します。次の ItemsPage.xaml の XAML は、プロパティを設定し **ItemClick** イベントを追加した **GridView** を示しています。

XAML (ItemsPage.xaml、黄色マーカー部を追記する)

```
<GridView
    x:Name="itemGridView"
    AutomationProperties.AutomationId="ItemsGridView"
    AutomationProperties.Name="Items"
    Margin="116,0,116,46"
    TabIndex="1"
    Grid.Row="1"
    Margin="0,-4,0,0"
    Padding="116,0,116,46"
    ItemsSource="{Binding Source={StaticResource itemsViewSource}}"
    ItemTemplate="{StaticResource Standard250x250ItemTemplate}"
    SelectionMode="None"
    IsItemClickEnabled="True"
    ItemClick="ItemView_ItemClick" />
```

書式変更: フォント : 太字, 蛍光ペン

また、アプリがスナップされている場合、項目ページにはグリッドの場所に表示される itemListView という名前のリスト ビューがあります。これについては、「さまざまなレイアウトへの対応」で詳しく説明します。ここでは、**GridView** に対して行った変更を **ListView** に対しても同様に行い、同じ動作を実現する必要があります。

XAML (ItemsPage.xaml、黄色マーカー部を追記する)

```
<ListView
    x:Name="itemListView"
    AutomationProperties.AutomationId="ItemsListView"
    AutomationProperties.Name="Items"
    Margin="10,0,0,60"
    TabIndex="1"
    Grid.Row="1"
    Visibility="Collapsed"
    Margin="0,-10,0,0"
    Padding="10,0,0,60"
    ItemsSource="{Binding Source={StaticResource itemsViewSource}}"
    ItemTemplate="{StaticResource Standard80ItemTemplate}"
    SelectionMode="None"
    IsItemClickEnabled="True"
    ItemClick="ItemView_ItemClick"/>
```

書式変更: フォント : 太字, 蛍光ペン

イベント ハンドラーを ItemsPage.xaml.cs 分離コード ページに追加します(XAML の ItemClick を右クリックして「イベントハンドラーへ移動」)。 ハンドラーでは、分割ページに移動し、選んだフィードのデータを渡します。

C# (ItemsPage.xaml.cs)

```
        private void ItemView_ItemClick(object sender, ItemClickEventArgs e)
        {
            // Navigate to the split page, configuring the new page
            // by passing the clicked item (FeedItem) as a navigation parameter
            this.Frame.Navigate(typeof(SplitPage), e.ClickedItem);
        }
```

書式変更: フォント : 太字, 蛍光ペン

書式変更: フォント : 太字

ページ間を移動するには、**Frame** コントロールの **Navigate** メソッド、**GoForward** メソッド、**GoBack** メソッドを使うことができます。Microsoft Visual Studio ページ テンプレートには、前に戻るための [戻る] ボタンがあります。BackButton の **Click** イベントによって **Frame.GoBack** メソッドが呼び出されます(ここで実際に動かして、BackButton を試してみてください)。

Navigate(TypeName, Object) メソッドを使うと、ページを移動してデータ オブジェクトを新しいページに渡すことができます。ここでは、ページ間でデータを渡すためにこのメソッドを使います。1 つ目のパラメーターの typeof(SplitPage) は、移動先のページの Type を示します。2 つ目のパラメーターは、移動先のページに渡すデータ オブジェクトを示します。この場合は、クリックされた項目を渡します。

F5 キーを押して、アプリをビルドし、実行します。ここで項目をクリックすると、選んだ項目の代わりに分割ページが読み込まれます。ただし、まだデータに接続していないので、分割ページは空のままです。

SplitPage.xaml.cs 分離コード ページでは、項目ページから渡された FeedData オブジェクトに対する処理を行う必要があります。そのためには、もう一度 LoadState メソッドを上書きします。このメソッドは既にページ テンプレート コードに追加されているため、必要な作業は、データをフックするように変更を加えるだけです。("ページ状態管理" 領域を展開して LoadState メソッドを表示することが必要な場合があります)。テンプレート ページには、データをフックできる DefaultViewModel という名前の組み込みのビュー モデルが含まれています。項目ページから渡されたデータ オブジェクトは、navigationParameter に含まれています。 これを FeedData オブジェクトにキャストします。次に、フィード データを Feed キーで DefaultViewModel に追加し、FeedData.Items プロパティを Items キーで DefaultViewModel に追加します。更新後の LoadState メソッドは次のようになります。

C# (SplitPage.xaml.cs、黄色マーカー部を追加)

```
        protected override void LoadState(Object navigationParameter,
Dictionary<String, Object> pageState)
        {
            // TODO: Assign a bindable group to this.DefaultViewModel["Group"]
            // TODO: Assign a collection of bindable items to
this.DefaultViewModel["Items"]
            FeedData feedData = navigationParameter as FeedData;
            if (feedData != null)
            {
                this.DefaultViewModel["Feed"] = feedData;
                this.DefaultViewModel["Items"] = feedData.Items;
            }

            if (pageState == null)
            {
                // When this is a new page, select the first item automatically unless
logical page
                // navigation is being used (see the logical page navigation #region
below.)
                if (!this.UsingLogicalPageNavigation() && this.itemsViewSource.View !=
null)
```

書式変更: フォント : 太字

書式変更: フォント : 太字, 蛍光ペン

書式変更: フォント : 太字

```
                {
                    this.itemsViewSource.View.MoveCurrentToFirst();
                }
            }
            else
            {
                // Restore the previously saved state associated with this page
                if (pageState.ContainsKey("SelectedItem") &&
this.itemsViewSource.View != null)
                {
                    // TODO: Invoke this.itemsViewSource.View.MoveCurrentTo() with the
selected
                    //       item as specified by the value of
pageState["SelectedItem"]
                }
            }
        }
```

F5 キーを押して、アプリをビルドし、実行します。分割ページに移動するとページが表示されるので、ブログ投稿の一覧から選ぶことができます。ただし、ページ タイトルはまだ空のままです。今からこれを修正します。

Microsoft Visual Studio ページ テンプレートでは、TODO コメントに~~よって~~、Group キーで DefaultViewModel にデータ オブジェクトを追加~~する位置を指示します~~しろと書いてあります(そうしておけば、ページ タイトルが表示された)。ブログ フィードはブログ投稿のグループであると考えることもできますが、Group の代わりに Feed キーを使うとコードが読みやすくなります。Group キーの代わりに Feed キーを使ったので、Group ではなく Feed プロパティにバインドするように、ページ タイトルのバインディングを変更する必要があります。SplitPage.xaml で、pageTitle という名前の **TextBlock** の **Text** バインディングを変更して、Text="{Binding Feed.Title}" のように Feed.Title にバインドします。

XAML (SplitPage.xaml)

```
<TextBlock x:Name="pageTitle" Grid.Column="1" Text="{Binding Feed.Title}"
           Style="{StaticResource PageHeaderTextStyle}"/>
```

書式変更: フォント : 太字, 蛍光ペン

F5 キーを押して、アプリをビルドし、実行します。これで、分割ページに移動すると、ページ タイトルが表示されます。

アプリに付け加えた新しいページへの機能の追加を完成させるには、もう少し変更を加える必要があります。次のコードをアプリに追加し終えたら、アプリのスタイリングとアニメーションに進み、目的の外観を作ることができます。

ItemsPage.xaml では、ページ タイトルが AppName キーで静的リソースにバインドされています。**Windows チーム ブログ**に対するこのリソースのテキストを次のように更新します。

XAML (ItemsPage.xaml)

```
<x:String x:Key="AppName">Windows Team Blogs</x:String>
```

書式変更: フォント : 太字, 蛍光ペン

SplitPage.xaml で、titlePanel という名前のグリッドを変更して、2 列に拡張します。

XAML (SplitPage.xaml)

```
<!-- Back button and page title [戻る] ボタンおよびページ タイトル -->
<Grid x:Name="titlePanel" Grid.ColumnSpan="2">
```

書式変更: フォント : 太字, 蛍光ペン

また、SplitPage.xaml では、選んだブログ投稿のタイトルと投稿内容の表示に使うレイアウトを変更する必要があります。これを行うには、itemDetail という名前の ScrollViewer をこの **ScrollViewer** レイアウトに置き換えます。

XAML (SplitPage.xaml)

```
        <!-- Details for selected item 選択したアイテムの詳細 -->
        <ScrollViewer
            x:Name="itemDetail"
            AutomationProperties.AutomationId="ItemDetailScrollViewer"
            Grid.Column="1"
            Grid.Row="1"
            Padding="70,0,120,0"
            DataContext="{Binding SelectedItem, ElementName=itemListView}"
            Style="{StaticResource VerticalScrollViewerStyle}">

            <Grid x:Name="itemDetailGrid">
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto"/>
                    <RowDefinition Height="*"/>
                </Grid.RowDefinitions>

                <TextBlock x:Name="itemTitle" Text="{Binding Title}"
                           Style="{StaticResource SubheaderTextStyle}"/>
                <Border x:Name="contentViewBorder" BorderBrush="Gray"
BorderThickness="2"
                        Grid.Row="1" Margin="0,15,0,20">
                    <Grid>
                        <WebView x:Name="contentView" />
                        <Rectangle x:Name="contentViewRect" />
                    </Grid>
                </Border>
            </Grid>
```

書式変更: フォント : 太字, 蛍光ペン

書式変更: フォント : 太字, 蛍光ペン

書式変更: フォント : 太字

```
        </ScrollViewer>
```

SplitPage.xaml.cs では、次のように ItemListView_SelectionChanged イベント ハンドラーにコードを追加して、選んだブログ投稿の内容を **WebView** に設定します。このイベント ハンドラーは、コードの**論理ページ ナビゲーション**領域 "Logical page navigation" リージョンにあります。

C# (SplitPage.xaml.cs)

```
        void ItemListView_SelectionChanged(object sender, SelectionChangedEventArgs e)
        {
            // Invalidate the view state when logical page navigation is in effect, as a change
            // in selection may cause a corresponding change in the current logical page.  When
            // an item is selected this has the effect of changing from displaying the item list
            // to showing the selected item's details.  When the selection is cleared this has the
            // opposite effect.
            if (this.UsingLogicalPageNavigation())
            {
                this.InvalidateVisualState();
            }

            // Add this code to populate the web view
            //  with the content of the selected blog post.
            Selector list = (Selector)sender;
            FeedItem selectedItem = (FeedItem)list.SelectedItem;
            if (selectedItem != null)
            {
                this.contentView.NavigateToString(selectedItem.Content);
            }
            else
            {
                this.contentView.NavigateToString("");
            }
        }
```

書式変更: フォント : 太字, 蛍光ペン

書式変更: フォント : 太字

DetailPage.xaml では、ブログ投稿のタイトルにタイトル テキストをバインドして、ブログ ページを表示する **WebView** コントロールを追加する必要があります。これを行うには、戻るボタンとページ タイトルを含む **Grid** をこの **Grid** と **WebView** に置き換えます。

XAML (DetailPage.xaml)

```
        <!-- Back button and page title -->
        <Grid>
```

```
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto"/>
                <ColumnDefinition Width="*"/>
            </Grid.ColumnDefinitions>
            <Button x:Name="backButton" Click="GoBack"
                    IsEnabled="{Binding Frame.CanGoBack, ElementName=pageRoot}"
                    Style="{StaticResource BackButtonStyle}"/>
            <TextBlock x:Name="pageTitle" Grid.Column="1"
                       Text="{Binding Title}"
                       Style="{StaticResource PageHeaderTextStyle}"/>
        </Grid>
        <Border x:Name="contentViewBorder" BorderBrush="Gray" BorderThickness="2"
                Grid.Row="1" Margin="120,15,20,20">
            <WebView x:Name="contentView" />
        </Border>
```

書式変更: フォント : 太字, 蛍光ペン

書式変更: フォント : 太字, 蛍光ペン

書式変更: フォント : 太字

DetailPage.xaml.cs では、LoadState メソッドにコードを追加してブログ投稿に移動するように上書きし、ページの **DataContext** を設定します。更新後の LoadState メソッドは次のようになります。

C# (DetailPage.xaml.cs)

```
        protected override void LoadState(Object navigationParameter,
Dictionary<String, Object> pageState)
        {
            // Add this code to navigate the web view to the selected blog post.
            FeedItem feedItem = navigationParameter as FeedItem;
            if (feedItem != null)
            {
                this.contentView.Navigate(feedItem.Link);
                this.DataContext = feedItem;
            }
        }
```

書式変更: フォント : 太字, 蛍光ペン

書式変更: フォント : 太字

F5 キーを押して、アプリをビルドし、実行します。これですべてのデータがフックされましたが、UI はあまり洗練されていません。また、作成した詳細ページに移動する手段もありません。次は、詳細ページにナビゲーションを追加します。

アプリ バーの追加

このブログ リーダー アプリのナビゲーションのほとんどは、ユーザーが UI で項目を選んだときに発生します。ただし、分割ページには、ユーザーがブログ投稿の詳細ビューを表示するための手段を用意しておく必要があります。ページ上のどこかにボタンを置くこともできますが、そうした場合、アプリの基本的な機能である "読む" 機能が犠牲になります。ここでは、ユーザーが必要とするまで非表示になるアプリ バーにボタンを置きます。

アプリ バーは UI の 1 つで、既定で非表示になっています。ユーザーが画面の端をスワイプするか、アプリを操作することで、開いたり閉じたりできます。アプリ バーでは、ナビゲーション、コマンド、ツールがユーザーに表示されます。アプリ バーは、ページの上部と下部のどちらか、またはその両方に表示できます。ナビゲーションを上部のアプリ バーに、ツールとコマンドを下部のアプリ バーにそれぞれ配置することをお勧めします。

XAML でアプリ バーを追加するには、AppBar コントロールを Page の TopAppBar プロパティまたは BottomAppBar プロパティに割り当てます。 上部のアプリ バーに、詳細ページに移動するボタンを追加しましょう。StandardStyles.xaml ファイルには、一般的なシナリオに対応したさまざまなアプリ バー ボタンのスタイルが含まれています。今回のボタンのスタイルを作成するにあたっては、これらのスタイルを使います。SplitPage.xaml の Page.Resources セクションにスタイルを記述し、次に示すように、リソース セクションの直後に **Page.TopAppBar** xaml を追加します。

XAML (SplitPage.xaml にナビゲーション バー(必ず上端)を追加する)

```
<Page.Resources>
    <!-- このページで表示されるアイテムのコレクション -->
    <CollectionViewSource
            x:Name="itemsViewSource"
            Source="{Binding Items}"/>

...
    <Style x:Key="WebViewAppBarButtonStyle" TargetType="Button"
           BasedOn="{StaticResource AppBarButtonStyle}">
        <Setter Property="AutomationProperties.AutomationId"
Value="WebViewAppBarButton"/>
        <Setter Property="AutomationProperties.Name" Value="View Web Page"/>
        <Setter Property="Content" Value="&#xE12B;"/>
    </Style>
</Page.Resources>

<Page.TopAppBar>
    <AppBar Padding="10,0,10,0">
        <Grid>
```

書式変更: フォント : 太字, 蛍光ペン

書式変更: フォント : 太字

書式変更: フォント : 太字, 蛍光ペン

```
            <Button Click="ViewDetail_Click" HorizontalAlignment="Right"
                    Style="{StaticResource WebViewAppBarButtonStyle}"/>
        </Grid>
    </AppBar>
</Page.TopAppBar>
```

アプリ バーの表示と非表示を切り替える方法を設定するには、IsSticky プロパティと IsOpen プロパティを使います。また、Opened イベントと Closed イベントを処理して、アプリ バーの開閉に応答することもできます。

詳細ページへのナビゲーションを処理するには、このコードを SplitPage.xaml.cs に追加します。

C# (SplitPage.xaml.cs)

```
private void ViewDetail_Click(object sender, RoutedEventArgs e)
{
    FeedItem selectedItem = this.itemListView.SelectedItem as FeedItem;
    if (selectedItem != null && this.Frame != null)
    {
        this.Frame.Navigate(typeof(DetailPage), selectedItem);
    }
}
```

書式変更: フォント : 太字

書式変更: フォント : 太字, 蛍光ペン

F5 キーを押して、アプリをビルドし、実行します。ブログをクリックして分割ページに移動し、読みたいブログ投稿を選びます。ヘッダーを右クリックしてアプリ バーを表示し、**[Show Web View](Web ビューの表示)** をクリックして詳細ページに移動します。

アニメーションと切り替えの追加

アニメーションと言うと、画面上をオブジェクトが飛び回るようなものをイメージしがちです。しかし XAML では、アニメーションは基本的にオブジェクトのプロパティの値を変更する 1 つの方法に過ぎません。これは飛び回るボールよりもはるかに役立ちます。このブログ リーダー アプリでは、アニメーションを使って UI をさまざまなレイアウトや方向に適合させます。詳しい方法については次のセクションで説明しますが、ここではまず、アニメーションの原理について理解する必要があります。

アニメーションを使うには、それを Storyboard の内部に置きます。**Storyboard** が実行されると、アニメーションの指定に沿ってプロパティが変化します。**Storyboard** の中には、1 つまたは複数

のアニメーションを含めることができます。それぞれのアニメーションによって、ターゲット オブジェクト、そのオブジェクトで変更するプロパティ、そのプロパティの新しい値が指定されます。

このブログ リーダー アプリには、itemListView という名前の **ListView** があります。次に示すアニメーションでは、**Storyboard** の実行時に itemListView の **Visibility** プロパティを **Visible** に変更します。

XAML (ItemPage.xaml の 113 行目付近)

```
<Storyboard>
    <ObjectAnimationUsingKeyFrames Storyboard.TargetName="itemListView"
                                   Storyboard.TargetProperty="Visibility" >
        <DiscreteObjectKeyFrame KeyTime="0" Value="Visible"/>
    </ObjectAnimationUsingKeyFrames>
</Storyboard>
```

テーマ アニメーションの追加

アプリを再び実行して、ページが読み込まれると、UI 要素がスムーズにスライドします。例外は、追加した **WebView** コントロールです。これは単に表示されただけです。Windows 8 では、UI にアニメーションと切り替えを使ってユーザー エクスペリエンスを高めています。ここでは、Windows 8 の特性に合うような操作性をアプリに組み込むことにします。Windows 8 で使われるものと同じテーマ アニメーションとテーマ切り替えが組み込まれているので、アプリでもそれを使うことができます。これらは、**Windows.UI.Xaml.Media.Animation** 名前空間に含まれています。

*テーマ アニメーション*とは、あらかじめ構成されたアニメーションであり、**Storyboard** 上に置くことができます。 **PopInThemeAnimation** により、ページが読み込まれたときに Web ビューが右から左にスライドします。**FromHorizontalOffset** プロパティの値を増やすと、効果がより極端になります。ここでは、**PopInThemeAnimation** を **Storyboard** 内に置き、これを DetailPage.xaml のリソースにします。アニメーションのターゲットに Web コンテンツを囲む **Border** を設定します。これによって、**Border** とその中のすべてのコンテンツにアニメーションが適用されます。

XAML (SplitPage.xaml)

```
<Page.Resources>
    ・・・
    </Style>
```

```
    <Storyboard x:Name="PopInStoryboard">
        <PopInThemeAnimation  Storyboard.TargetName="contentViewBorder"
                              FromHorizontalOffset="400"/>
    </Storyboard>
</Page.Resources>
```

書式変更: フォント : 太字
書式変更: フォント : 太字, 蛍光ペン
書式変更: フォント : 太字

ユーザーが詳細ページを開いたときに Border にポップイン アニメーションが適用されるように、分離コード ページの LoadState メソッドで Storyboard を開始します。更新後の LoadState メソッドは次のようになります。

C# (SplitPage.xaml.cs)

```
        protected override void LoadState(Object navigationParameter,
Dictionary<String, Object> pageState)
        {
            // Run the PopInThemeAnimation
            Windows.UI.Xaml.Media.Animation.Storyboard sb =
                this.FindName("PopInStoryboard") as
Windows.UI.Xaml.Media.Animation.Storyboard;
            if (sb != null) sb.Begin();

            // TODO: Assign a bindable group to this.DefaultViewModel["Group"]
            // TODO: Assign a collection of bindable items to
this.DefaultViewModel["Items"]
            FeedData feedData = navigationParameter as FeedData;
            if (feedData != null)
            {
                this.DefaultViewModel["Feed"] = feedData;
                this.DefaultViewModel["Items"] = feedData.Items;
            }

            if (pageState == null)
            {
                // When this is a new page, select the first item automatically unless
logical page
                // navigation is being used (see the logical page navigation #region
below.)
                if (!this.UsingLogicalPageNavigation() && this.itemsViewSource.View !=
null)
                {
                    this.itemsViewSource.View.MoveCurrentToFirst();
                }
            }
            else
            {
                // Restore the previously saved state associated with this page
```

書式変更: フォント : 太字
書式変更: フォント : 太字, 蛍光ペン
書式変更: フォント : 太字

```
                if (pageState.ContainsKey("SelectedItem") &&
this.itemsViewSource.View != null)
                {
                    // TODO: Invoke this.itemsViewSource.View.MoveCurrentTo() with the
selected
                    //       item as specified by the value of
pageState["SelectedItem"]
                }
            }
        }
```

F5 キーを押して、アプリをビルドし、実行します。分割ページが読み込まれると、**WebView** コントロールは他の UI 要素と共にスライドします。

詳しい情報と、テーマ アニメーションおよびテーマ切り替えの一覧については、アニメーションに関するクイック スタート トピックをご覧ください。

スタイルを使った外観の統一

ブログ リーダー アプリの外観を、Windows チーム ブログ Web サイトと同様のデザインに合わせることにします。ユーザーが Web サイトとアプリの間を移動しても違和感を感じないようにすることが目的です。現在の Windows Metro スタイル UI の黒を基調とした既定のテーマは、Windows チーム ブログ Web サイトにはあまり合いません。このことは、**WebView** に実際のブログ ページが読み込まれる詳細ページを見ると、はっきりとわかります。次の図をご覧ください。

アプリに一貫した外観を適用し、必要に応じて更新できるようにするには、ブラシとスタイルを使います。**Brush** を使うと、1 か所で外観を定義しておき、それを必要な場所で適用することができます。**Style** を使うと、コントロールのプロパティに値を設定し、その設定をアプリ全体で再利用することができます。

詳しい説明の前に、このアプリでブラシを使ってページの背景色を設定する方法を見てみましょう。アプリの各ページには、ページの背景色を定義するために **Background** プロパティが設定されたルートの **Grid** があります。各ページの背景は、次のように個別に設定できます。

XAML

```
<Grid Background="Blue">
```

お勧めは、**Brush** をリソースとして定義してこれをすべてのページの背景色の定義に使う方法です。オブジェクトと値をリソースとして定義し、再利用できるようにします。オブジェクトまた

は値をリソースとして使うには、**x:Key** 属性を設定する必要があります。このキーを使って、XAML からリソースを参照します。ここでは、システム定義の **SolidColorBrush** である ApplicationPageBackgroundThemeBrush キーを使ってリソースに背景が設定されています。

XAML

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
```

Grid の **Background** プロパティを設定してもよいですが、目的の外観を作るには一般に複数のプロパティを設定する必要があります。プロパティの設定を好きな数だけまとめて **Style** にグループ化し、**Style** をコントロールに適用できます。

リソースは、個々のページの XAML ファイル、App.xaml ファイル、StandardStyles.xaml などの別個のリソース ディクショナリ XAML ファイルに定義できます。リソースを定義する場所は、リソースが使われる範囲によって決まります。Microsoft Visual Studio ~~で~~は、StandardStyles.xaml ファイルをプロジェクト テンプレートの一部として作成し、[**共通 Common**] フォルダーに配置します。このファイルは、Visual Studio ページ テンプレートで使われる値、スタイル、データ テンプレートを含むリソース ディクショナリです。リソース ディクショナリ XAML ファイルはアプリ間で共有できます。また、単一のアプリで複数のリソース ディクショナリをマージすることも可能です。

このブログ リーダー アプリでは App.xaml にリソースを定義して、アプリ全体で使えるようにします。また、個々のページの XAML ファイルでも、いくつかのリソースが定義されています。このようなリソースは定義元のページでしか利用できません。App.xaml とページ内の両方で同じキーを持つリソースが定義されている場合、ページ内のリソースが App.xaml 内のリソースよりも優先されます。同様に、App.xaml に定義されているリソースは、別個のリソース ディクショナリ ファイルに同じキーを使って定義されたリソースよりも優先されます。詳しくは、「クイック スタート: コントロールのスタイル」をご覧ください。

次に、このアプリにおける **Style** の使用例を紹介します。テンプレート ページの外観は LayoutRootStyle キーを持つスタイルによって定義されています。**Style** の定義は StandardStyles.xaml ファイルにあります。

XAML (StandardStyles.xaml)

```
<Grid Style="{StaticResource LayoutRootStyle}">
```

XAML

```
    <!-- Page layout roots typically use entrance animations and a theme-appropriate background color -->
```

```
    <Style x:Key="LayoutRootStyle" TargetType="Panel">
        <Setter Property="Background" Value="{StaticResource
ApplicationPageBackgroundThemeBrush}"/>
        <Setter Property="ChildrenTransitions">
            <Setter.Value>
                <TransitionCollection>
                    <EntranceThemeTransition/>
                </TransitionCollection>
            </Setter.Value>
        </Setter>
    </Style>
```

Style の定義では、1 つの **TargetType** 属性と、1 つ以上の **Setter** が必要になります。**TargetType** は、**Style** が適用される種類を指定する文字列 (この場合は **Panel**) に設定します。**TargetType** 属性の指定内容と異なるコントロールに **Style** を適用しようとすると、例外が発生します。それぞれの **Setter** 要素に、**Property** および **Value** が必要です。この 2 つのプロパティは、それぞれ、その設定が適用されるコントロールのプロパティと、そのプロパティに対して設定される値を指定します。

ページの **Background** を変更するには、ApplicationPageBackgroundThemeBrush を独自のカスタム ブラシで置き換える必要があります。カスタム ブラシでは **Color** に #FF0A2562 を設定します。美しい青色で、http://windowsteamblog.com の配色ともよく合います。システムのテーマのブラシを置き換えるには、LayoutRootStyle に基づいた新しい **Style** を作り、そこで **Background** プロパティを変更します。次に、レイアウト ルートの新しいスタイルを定義する方法を示します。

XAML

```
                    <SolidColorBrush x:Key="WindowsBlogBackgroundBrush"
Color="#FF0A2562"/>
                    <Style x:Key="WindowsBlogLayoutRootStyle" TargetType="Panel"
BasedOn="{StaticResource LayoutRootStyle}">
                        <Setter Property="Background" Value="{StaticResource
WindowsBlogBackgroundBrush}"/>
                    </Style>
```

アプリ固有のその他のリソースを含む App.xaml 内の **ResourceDictionary** に、これらのブラシとスタイルの定義を配置します。

XAML

```
            <ResourceDictionary.MergedDictionaries>
                <!--
```

```
                Styles that define common aspects of the platform look and feel
                Required by Visual Studio project and item templates
            -->
            <ResourceDictionary Source="Common/StandardStyles.xaml"/>

            <ResourceDictionary>
                <local:FeedDataSource x:Key="feedDataSource"/>
                <local:DateConverter x:Key="dateConverter" />

                <SolidColorBrush x:Key="WindowsBlogBackgroundBrush"
                                 Color="#FF0A2562"/>
                <Style x:Key="WindowsBlogLayoutRootStyle"
                       TargetType="Panel"
                       BasedOn="{StaticResource LayoutRootStyle}">
                    <Setter Property="Background"
                            Value="{StaticResource WindowsBlogBackgroundBrush}"/>
                </Style>
            </ResourceDictionary>
        </ResourceDictionary.MergedDictionaries>
```

重要 このスタイルは StandardStyles.xaml のシステム スタイルに基づいているため、**MergedDictionaries** では StandardStyles.xaml を含む **ResourceDictionary** をアプリの **ResourceDictionary** の前に宣言する必要があります。こうしないと、XAML パーサーでは、スタイルの基になる LayoutRootStyle を見つけられません。

BasedOn="{StaticResource LayoutRootStyle}" によって、明示的に設定しないプロパティをすべて LayoutRootStyle から新しい **Style** に継承するように指定しています。新しい **Style** は既定のスタイルと似ていますが、背景は青色です。次に示すように、これをすべてのページで使うことができます。

XAML

```
<Grid Style="{StaticResource WindowsBlogLayoutRootStyle}">
```

ItemsPage.xaml、SplitPage.xaml、DetailPage.xaml でルート **Grid** を更新して、WindowsBlogLayoutRootStyle を使います。F5 キーを押し、アプリをビルドして実行します。青いページが表示されます。

アプリを Windows チーム ブログの Web サイトに適した外観にするために、**Brush** と **Style** に加えて、カスタム データ テンプレートも使います。データ テンプレートについては、「データの表示」で説明しました。ここでは、データ テンプレートとスタイルを追加してアプリの外観をカスタマイズする方法を示します。

App.xaml で、日付を表示する四角のブロックを定義する **ControlTemplate** を追加します。
App.xaml にこれを定義し、ItemsPage.xaml と SplitPage.xaml の両方で使えるようにします。

XAML

```
    <Application.Resources>
        <ResourceDictionary>

                                                                    ...
            <ControlTemplate x:Key="DateBlockTemplate">
                <Canvas Height="86" Width="86"  Margin="8,8,0,8"
HorizontalAlignment="Left" VerticalAlignment="Top">
                    <TextBlock TextTrimming="WordEllipsis" TextWrapping="NoWrap"
                     Width="Auto" Height="Auto" Margin="8,0,4,0" FontSize="32"
FontWeight="Bold">
                        <TextBlock.Text>
                            <Binding Path="PubDate" Converter="{StaticResource
dateConverter}" ConverterParameter="month"  />
                        </TextBlock.Text>
                    </TextBlock>

                    <TextBlock TextTrimming="WordEllipsis" TextWrapping="Wrap"
                     Width="40" Height="Auto" Margin="8,0,0,0" FontSize="34"
FontWeight="Bold" Canvas.Top="36">
                        <TextBlock.Text>
                            <Binding Path="PubDate" Converter="{StaticResource
dateConverter}" ConverterParameter="day"  />
                        </TextBlock.Text>
                    </TextBlock>
                    <Line Stroke="White" StrokeThickness="2" X1="54" Y1="46" X2="54"
Y2="80" />

                    <TextBlock TextWrapping="Wrap"
                     Width="20" Height="Auto" FontSize="{StaticResource
ContentFontSize}" Canvas.Top="42" Canvas.Left="60">
                        <TextBlock.Text>
                            <Binding Path="PubDate" Converter="{StaticResource
dateConverter}" ConverterParameter="year"  />
                        </TextBlock.Text>
                    </TextBlock>
                </Canvas>
            </ControlTemplate>
            ...
    </Application.Resources>
```

ItemsPage.xaml では、これらのリソースを追加して既定のビューのグリッド項目の外観を定義します。

XAML

```
<Page.Resources>

...
    <!-- light blue -->
    <SolidColorBrush x:Key="BlockBackgroundBrush" Color="#FF557EB9"/>

    <!-- Grid Styles -->
    <Style x:Key="GridTitleTextStyle" TargetType="TextBlock" BasedOn="{StaticResource
BasicTextStyle}">
        <Setter Property="FontSize" Value="26.667"/>
        <Setter Property="Margin" Value="12,0,12,2"/>
    </Style>

    <Style x:Key="GridDescriptionTextStyle" TargetType="TextBlock"
BasedOn="{StaticResource BasicTextStyle}">
        <Setter Property="VerticalAlignment" Value="Bottom"/>
        <Setter Property="Margin" Value="12,0,12,60"/>
    </Style>

    <DataTemplate x:Key="DefaultGridItemTemplate">
        <Grid HorizontalAlignment="Left" Width="250" Height="250">
            <Border Background="{StaticResource BlockBackgroundBrush}" />
            <TextBlock Text="{Binding Title}" Style="{StaticResource
GridTitleTextStyle}"/>
            <TextBlock Text="{Binding Description}" Style="{StaticResource
GridDescriptionTextStyle}" />
            <StackPanel VerticalAlignment="Bottom" Orientation="Horizontal"
                        Background="{StaticResource
ListViewItemOverlayBackgroundThemeBrush}">
                <TextBlock Text="Last Updated" Margin="12,4,0,8" Height="42"/>
                <TextBlock Text="{Binding PubDate, Converter={StaticResource
dateConverter}}" Margin="12,4,12,8" />
            </StackPanel>
        </Grid>
    </DataTemplate>
</Page.Resources>
```

また、ItemsPage.xaml では、itemGridView の **ItemTemplate** プロパティを更新して、StandardStyles.xaml に定義された既定のテンプレートである Standard250x250ItemTemplate の代わりに、DefaultGridItemTemplate リソースを使う必要があります。itemGridView の更新後の XAML は次のようになります。

XAML

```
            <GridView
                x:Name="itemGridView"
```

```
                AutomationProperties.AutomationId="ItemsGridView"
                AutomationProperties.Name="Items"
                Margin="116,0,116,46"
                ItemsSource="{Binding Source={StaticResource itemsViewSource}}"

                ItemTemplate="{StaticResource DefaultGridItemTemplate}"

                SelectionMode="None"
                IsItemClickEnabled="True"
                ItemClick="ItemView_ItemClick"/>
```

SplitPage.xaml では、これらのリソースを追加して既定のビューのリスト項目の外観を定義します。

XAML

```
<Page.Resources>
                            ...
    <!-- green -->
    <SolidColorBrush x:Key="BlockBackgroundBrush" Color="#FF6BBD46"/>

    <DataTemplate x:Key="DefaultListItemTemplate">
        <Grid HorizontalAlignment="Stretch" Width="Auto" Height="110"
Margin="10,10,10,0">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto"/>
                <ColumnDefinition Width="*"/>
            </Grid.ColumnDefinitions>
            <!-- Green date block -->
            <Border Background="{StaticResource BlockBackgroundBrush}" Width="110"
Height="110" />
            <ContentControl Template="{StaticResource DateBlockTemplate}" />
            <StackPanel Grid.Column="1"  HorizontalAlignment="Left" Margin="12,8,0,0">
                <TextBlock Text="{Binding Title}" FontSize="26.667" TextWrapping="Wrap"
                           MaxHeight="72" Foreground="#FFFE5815" />
                <TextBlock Text="{Binding Author}" FontSize="18.667" />
            </StackPanel>
        </Grid>
    </DataTemplate>
    ...
</Page.Resources>
```

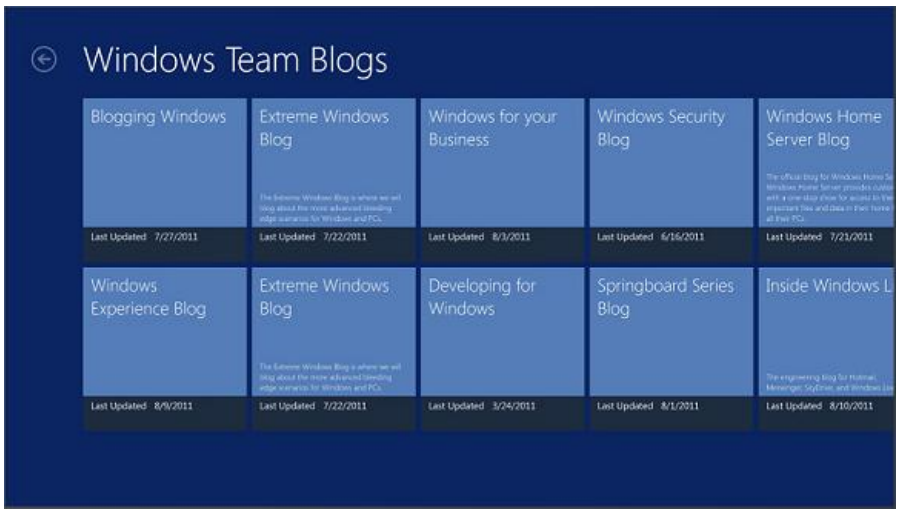
また、SplitPage.xaml では、itemListView の **ItemTemplate** プロパティを更新して、既定のテンプレートである Standard130ItemTemplate の代わりに、DefaultListItemTemplate リソースを使う必要があります。itemListView の更新後の XAML は次のようになります。

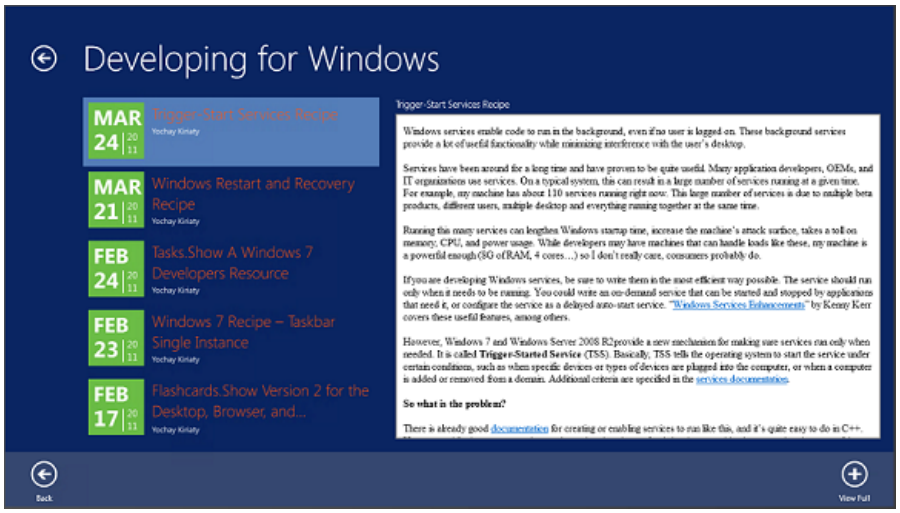
XAML

```
        <ListView
            x:Name="itemListView"
            AutomationProperties.AutomationId="ItemsListView"
            AutomationProperties.Name="Items"
            Margin="120,0,0,60"
            ItemsSource="{Binding Source={StaticResource itemsViewSource}}"
            SelectionChanged="ItemListView_SelectionChanged"

            ItemTemplate="{StaticResource DefaultListItemTemplate}"/>
```

アプリにこのスタイルが適用されると、Windows チーム ブログ Web サイトの外観に適したデザインになります。





スタイルを使い、そのスタイルを他のスタイルに基づかせることで、アプリにさまざまな外観を簡単に定義して適用できるようになります。次のセクションでは、アニメーションとスタイルによって実行中のアプリをさまざまなレイアウトと方向に柔軟に対応させる方法を組み合わせます。

さまざまなレイアウトへの対応

通常、アプリは横方向の全画面で表示されることを想定して設計します。ただし、Metro スタイル UI はさまざまな方向とレイアウトに対応できることが必要です。特に、横方向と縦方向の両方をサポートする必要があります。横方向の場合、*全画面レイアウト*、*ページ横幅に合わせたレイアウト*、*スナップされたレイアウト*をサポートする必要があります。既に述べたように、空のテンプレートからこのブログ リーダー ページを作った場合、縦方向ではうまく表示されません。このセクションでは、どの解像度や方向でもアプリをきれいに表示できる方法を紹介します。

書式変更: 蛍光ペン

注 さまざまな表示の向きや解像度でアプリをテストするには、アプリをシミュレーターで実行できます。Metro スタイル アプリをシミュレーターで実行するには、[標準] ツール バーの **[デバッグの開始]** の横のドロップダウン リストから、**[シミュレーター]** を選びます。シミュレーターについて詳しくは、Visual Studio からの Metro スタイル アプリの実行に関するページをご覧ください。

Visual Studio テンプレートでは、含まれているコードでビュー状態の変更が処理されます。次の LayoutAwarePage.cs ファイルのコードでは、アプリの状態を XAML に定義された表示状態にマップしています。ページ レイアウトのロジックは提供されているため、各ページの表示状態に使うビューを用意するだけでかまいません。

XAML を使ってビューを切り替えるには、**VisualStateManger** を使って、アプリのさまざまな **VisualState** を定義します。ItemsPage.xaml に定義されている **VisualStateGroup** を次に示します。このグループには、FullScreenLandscape、Filled、FullScreenPortrait、Snapped という名前の 4 つの **VisualState** があります。同じ **VisualStateGroup** の異なる **VisualState** を同時に使うことはできません。 それぞれの **VisualState** に含まれているアニメーションで、UI に対して XAML で指定されているベースラインからの変更内容がアプリに指定されます。

XAML

```
<!--App Orientation States-->
  <VisualStateManager.VisualStateGroups>
    <VisualStateGroup x:Name="ApplicationViewStates">
      <VisualState x:Name="FullScreenLandscape" />
      <VisualState x:Name="Filled"> ... </VisualState>
      <VisualState x:Name="FullScreenPortrait"> ... </VisualState>
      <VisualState x:Name="Snapped"> ... </VisualState>
    </VisualStateGroup>
  </VisualStateManager.VisualStateGroups>
```

アプリが横方向の全画面表示の場合に、FullScreenLandscape 状態を使います。このビューには既定の UI が設定されているので、変更は必要なく、これは単なる空の **VisualState** です。

ユーザーが画面の片側に別のアプリをスナップしている場合に、Filled 状態を使います。この例では、項目ビュー ページが端に移動するだけで、変更は必要ありません。これも、単なる空の **VisualState** です。

アプリが横方向から縦方向に回転するときに、FullScreenPortrait 状態を使います。この表示状態では、2 つのアニメーションが実行されます。1 つは戻るボタンに使うスタイルを変更し、もう 1 つは全体がより画面にうまく合うように itemGridView の余白を変更します。

XAML

```
<!-- The entire page respects the narrower 100-pixel margin convention for portrait -->
<VisualState x:Name="FullScreenPortrait">
    <Storyboard>
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="backButton"
                                       Storyboard.TargetProperty="Style">
            <DiscreteObjectKeyFrame KeyTime="0" Value="{StaticResource
PortraitBackButtonStyle}"/>
        </ObjectAnimationUsingKeyFrames>
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="itemGridView"
                                       Storyboard.TargetProperty="Padding">
            <DiscreteObjectKeyFrame KeyTime="0" Value="96,0,86,56"/>
        </ObjectAnimationUsingKeyFrames>
    </Storyboard>
</VisualState>
```

ユーザーが 2 つのアプリを表示していて、このブログ リーダー アプリの方が狭く表示される場合に、Snapped 状態を使います。この状態では、ブログ リーダー アプリの幅がわずか 320 dip になるため、大幅なレイアウト変更が必要になります。項目ページの UI の XAML では **GridView** と **ListView** の両方が定義されており、データ コレクションにバインドされています。既定では、itemGridViewScroller が表示され、itemListViewScroller が折りたたまれます。Snapped 状態では、itemListViewScroller を折りたたむアニメーション、itemListViewScroller を表示するアニメーション、戻るボタンとページ タイトルの **Style** をより小さくするアニメーションの 4 つが実行されます。

XAML

```
<!--
    The back button and title have different styles when snapped, and the list
representation is substituted
    for the grid displayed in all other view states
```

```
-->
<VisualState x:Name="Snapped">
    <Storyboard>
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="backButton"
                                       Storyboard.TargetProperty="Style">
            <DiscreteObjectKeyFrame KeyTime="0"
                                    Value="{StaticResource SnappedBackButtonStyle}"/>
        </ObjectAnimationUsingKeyFrames>
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="pageTitle"
                                       Storyboard.TargetProperty="Style">
            <DiscreteObjectKeyFrame KeyTime="0"
                                    Value="{StaticResource
SnappedPageHeaderTextStyle}"/>
        </ObjectAnimationUsingKeyFrames>

        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="itemListScrollViewer"
                                       Storyboard.TargetProperty="Visibility">
            <DiscreteObjectKeyFrame KeyTime="0" Value="Visible"/>
        </ObjectAnimationUsingKeyFrames>
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="itemGridScrollViewer"
                                       Storyboard.TargetProperty="Visibility">
            <DiscreteObjectKeyFrame KeyTime="0" Value="Collapsed"/>
        </ObjectAnimationUsingKeyFrames>
    </Storyboard>
</VisualState>
```

このチュートリアルの「スタイルを使った外観の統一」では、スタイルとテンプレートを作成してアプリの外観をカスタマイズしました。既定の横方向ビューで使うスタイルとテンプレートは次のとおりです。また、カスタマイズした外観を別のビューでも維持する場合は、これらのビューのカスタム スタイルとカスタム テンプレートを作成する必要があります。

ItemsPage.xaml では、グリッド項目の新しいデータ テンプレートを作成しました。Snapped ビューでリスト項目を表示する場合にも、新しいデータ テンプレートを用意する必要があります。このテンプレートの名前を NarrowListItemTemplate とし、ItemsPage.xaml のリソース セクションの、DefaultGridItemTemplate リソースの直後に追加します。

XAML

```
<Page.Resources>
...
    <!-- Used in Snapped view -->
    <DataTemplate x:Key="NarrowListItemTemplate">
        <Grid Height="80">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto"/>
                <ColumnDefinition Width="*"/>
```

```
            </Grid.ColumnDefinitions>
            <Border Background="{StaticResource BlockBackgroundBrush}" Width="80"
Height="80" />
            <ContentControl Template="{StaticResource DateBlockTemplate}" Margin="-12,-
12,0,0"/>
            <StackPanel Grid.Column="1" HorizontalAlignment="Left" Margin="12,8,0,0">
                <TextBlock Text="{Binding Title}" MaxHeight="56" TextWrapping="Wrap"/>
            </StackPanel>
        </Grid>
    </DataTemplate>
</Page.Resources>
```

ListView に新しいデータ テンプレートを表示するには、itemListView の **ItemTemplate** プロパティを更新して、StandardStyles.xaml に定義された既定のテンプレートである Standard80ItemTemplate の代わりに、NarrowListItemTemplate リソースを使います。itemListView の更新後の XAML は次のようになります。

XAML

```
            <ListView
                x:Name="itemListView"
                AutomationProperties.AutomationId="ItemsListView"
                AutomationProperties.Name="Items"
                Margin="10,0,0,60"
                ItemsSource="{Binding Source={StaticResource itemsViewSource}}"

                ItemTemplate="{StaticResource NarrowListItemTemplate}"

                SelectionMode="None"
                IsItemClickEnabled="True"
                ItemClick="ItemView_ItemClick"/>
```

SplitPage.xaml では、Filled ビュー、Snapped ビュー、画面の幅が 1366 dip 未満の場合の FullScreenLandscape ビューで使われるものとよく似た **ListView** テンプレートを作成します。このテンプレートの名前も NarrowListItemTemplate とし、SplitPage.xaml のリソース セクションの、DefaultListItemTemplate リソースの直後に追加します。

XAML

```
<Page.Resources>
...
    <!-- Used in Filled and Snapped views -->
    <DataTemplate x:Key="NarrowListItemTemplate">
        <Grid Height="80">
            <Grid.ColumnDefinitions>
```

```
                <ColumnDefinition Width="Auto"/>
                <ColumnDefinition Width="*"/>
            </Grid.ColumnDefinitions>
            <Border Background="{StaticResource BlockBackgroundBrush}" Width="80"
Height="80"/>
            <ContentControl Template="{StaticResource DateBlockTemplate}" Margin="-12,-
12,0,0"/>
            <StackPanel Grid.Column="1" HorizontalAlignment="Left" Margin="12,8,0,0">
                <TextBlock Text="{Binding Title}" MaxHeight="56" Foreground="#FFFE5815"
TextWrapping="Wrap"/>
                <TextBlock Text="{Binding Author}" FontSize="12" />
            </StackPanel>
        </Grid>
    </DataTemplate>
...
</Page.Resources>
```

このデータ テンプレートを使うために、テンプレートを使う場所の表示状態を更新します。Snapped 表示状態と Filled 表示状態の XAML では、itemListView の **ItemTemplate** プロパティを対象とするアニメーションが実行されます。次に、既定の Standard80ItemTemplate リソースの代わりに NarrowListItemTemplate リソースを使うように、値を変更します。アニメーションの更新後の XAML は次のようになります。

XAML

```
<VisualState x:Name="Filled">
    <Storyboard>
    ....
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="itemListView"
Storyboard.TargetProperty="ItemTemplate">
            <DiscreteObjectKeyFrame KeyTime="0" Value="{StaticResource
NarrowListItemTemplate}"/>
        </ObjectAnimationUsingKeyFrames>
    ....
    </Storyboard>
</VisualState>
...
<VisualState x:Name="Snapped">
    <Storyboard>
    ....
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="itemListView"
Storyboard.TargetProperty="ItemTemplate">
            <DiscreteObjectKeyFrame KeyTime="0" Value="{StaticResource
NarrowListItemTemplate}"/>
        </ObjectAnimationUsingKeyFrames>
    ....
    </Storyboard>
```

```
</VisualState>
```

また、分割ページの項目の詳細セクションも **WebView** を使った独自の詳細セクションに置き換えました。この変更を行ったために、Snapped_Detail 表示状態のターゲット要素の一部のアニメーションは削除されています。これらのアニメーションは、この表示状態を使う際にエラーの原因になる可能性があるため、削除する必要があります。SplitPage.xaml で、Snapped_Detail 表示状態からこれらのアニメーションを削除します。

XAML

```
<VisualState x:Name="Snapped_Detail">
    <Storyboard>
    ...
        <!--<ObjectAnimationUsingKeyFrames Storyboard.TargetName="itemDetailTitlePanel"
Storyboard.TargetProperty="(Grid.Row)">
                <DiscreteObjectKeyFrame KeyTime="0" Value="0"/>
            </ObjectAnimationUsingKeyFrames>
            <ObjectAnimationUsingKeyFrames Storyboard.TargetName="itemDetailTitlePanel"
Storyboard.TargetProperty="(Grid.Column)">
                <DiscreteObjectKeyFrame KeyTime="0" Value="0"/>
            </ObjectAnimationUsingKeyFrames>-->
...
        <!--<ObjectAnimationUsingKeyFrames Storyboard.TargetName="itemSubtitle"
Storyboard.TargetProperty="Style">
                <DiscreteObjectKeyFrame KeyTime="0" Value="{StaticResource
CaptionTextStyle}"/>
            </ObjectAnimationUsingKeyFrames>-->
    </Storyboard>
</VisualState>
```

DetailPage.xaml で、利用可能なすべての領域を使うには、単に Snapped ビューの **WebView** の余白を調整するだけでかまいません。Snapped 表示状態の XAML に、次のように contentViewBorder の **Margin** プロパティの値を変更するアニメーションを追加します。

XAML

```
<VisualState x:Name="Snapped">
    <Storyboard>
    ...
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="contentViewBorder"
Storyboard.TargetProperty="Margin">
            <DiscreteObjectKeyFrame KeyTime="0" Value="20,5,20,20"/>
        </ObjectAnimationUsingKeyFrames>

    </Storyboard>
```

```
</VisualState>
```

スプラッシュ画面とロゴの追加

ユーザーがアプリから感じる第一印象は、ロゴとスプラッシュ画面によって決まります。ロゴは、Windows ストアのアプリのスタート画面に表示されます。スプラッシュ画面は、ユーザーがアプリを起動したときに短時間表示され、アプリがリソースを初期化している間に瞬間的なフィードバックをユーザーに示します。アプリの最初のページを表示する準備が完了すると、スプラッシュ画面が消えます。

スプラッシュ画面は、背景色と 620 x 300 ピクセルのイメージで構成されます。これを指定する値は、Package.appxmanifest ファイルで設定します。このファイルをダブルクリックして、マニフェスト エディターでファイルを開きます。マニフェスト エディターの **[アプリケーション UI]** タブで、スプラッシュ画面のイメージ ファイルのパスと背景色を設定します。プロジェクト テンプレートには、SplashScreen.png という名前の既定の空白イメージが用意されています。このイメージを、独自のスプラッシュ画面イメージに置き換えます。アプリの性格を明確に表現し、ユーザーがひとめで関心を持つようなイメージを使います。テンプレート ロゴ ファイルを独自のロゴ ファイルに置き換えてロゴを指定することもできます。このブログ リーダーのスプラッシュ画面は、次のようになります。

このブログ リーダーは基本的なスプラッシュ画面で十分ですが、**SplashScreen** クラスのプロパティとメソッドを使って、スプラッシュ画面を拡張することもできます。**SplashScreen** クラスを使うと、スプラッシュ画面の座標を取得し、それを使ってアプリの最初のページを配置することができます。さらに、スプラッシュ画面が消える時点を調べることができるため、アプリのコンテンツの導入アニメーションを開始するタイミングを把握できます。

次にすること

ここでは、Visual Studio Express 2012 RC for Windows 8 の組み込みページ テンプレートを使って完全なマルチ ページ アプリを作成する方法と、ページ間を移動してデータを渡す方法について説明しました。スタイルとテンプレートを使ってアプリを Windows チーム ブログの Web サイトの雰囲気に調和させる方法についても説明しました。さらに、テーマ アニメーション、アプリ バー、スプラッシュ画面を使って、アプリの外観を Windows 8 の雰囲気に合わせる方法についても説明しました。 最後に、アプリをさまざまなレイアウトや向きに対応させて常に最善の表示を得る方法について説明しました。

アプリ開発について詳しくは、Metro スタイル アプリの作成に関する学習リソースやリファレンス リソースの一覧をご覧ください。

C# または Visual Basic を使った Metro スタイル アプリのためのロードマップ。