Windows Phone 8: Treasure Hunt Application Example using NFC and Nokia Maps API

This article describes how to create a WP8-application in which the user hunts NFC tags as treasures using a Nokia Maps -map element as help.

Introduction

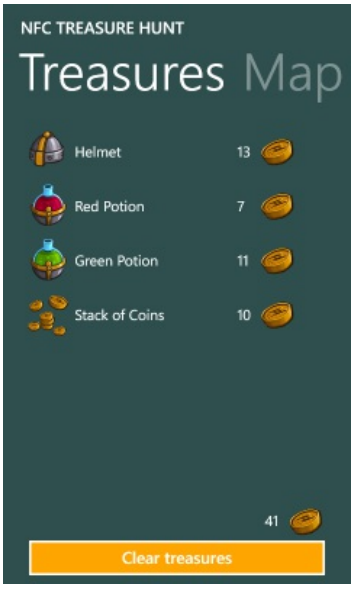
Windows Phone 8 comes with several new and handy features for both application developers and the devices' users. One of these features is the Near Field Communication (NFC) which allows the device user to read/send data to/from an NFC tag, sticker or device by placing the device 3-4 centimeters from the other tag or device. Small amounts of data, such as pieces of text or contact card information can be sent.

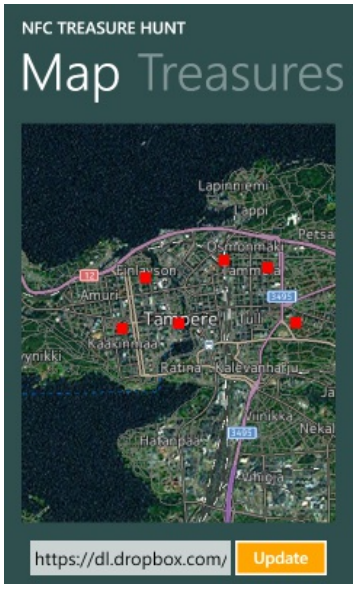
Windows Phone 8 also uses Nokia Maps as the primary map service instead of the old Bing Maps, and developers can now implement Nokia Maps to their applications with the new map API.

To showcase these features in practice, we are going to be making an example application that uses NFC to read different keywords from NFC tags and adds a treasure entry to a treasure list corresponding to different keywords. We implement a possibility to read map coordinates for the treasures from an online URL and show these coordinates as marks on a separate map page.

Please note that to test the NFC functionalities of the application, you must have a WP8 device since the emulator doesn't support NFC testing, and also a writeable NFC tag. You can of course test the other functionalities on the emulator if you disable or remove the NFC parts of the code.

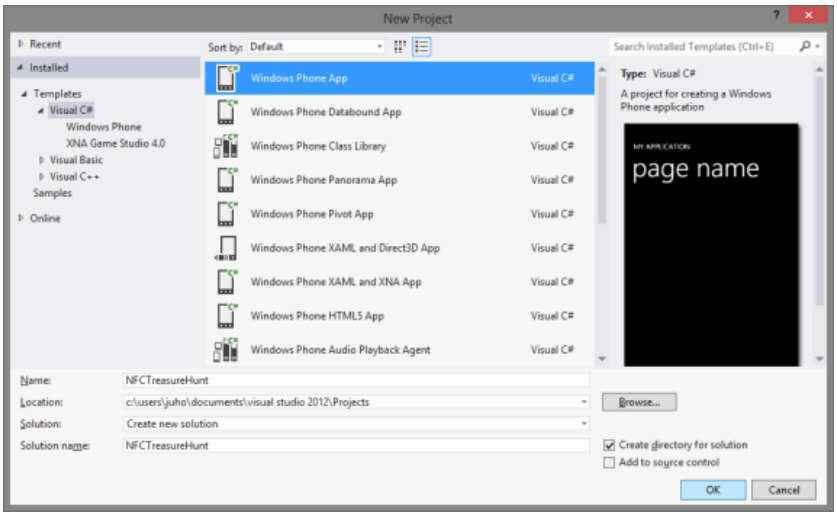
In the end of the example, the application is going to look like this:



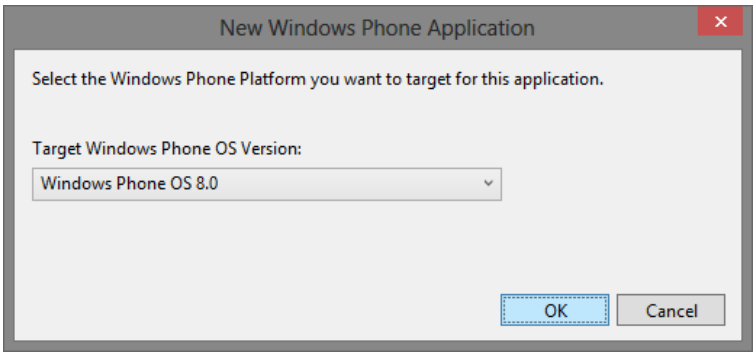


Part 1: Creating a New Visual Studio Project, Adding Project Assets and Modifying the XAML

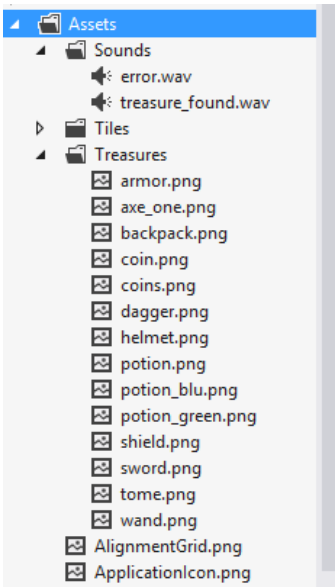
First up, let's create a new Visual Studio project from the basic Windows Phone App -template. Let's name the project NFCTreasureHunt. When Visual Studio prompts you to select an OS version to support, select Windows Phone 8.0.



(C) Copyright Nokia 2013. All rights reserved.



Next, download these resource files. In Visual Studio Solution Explorer, create two folders under the Assets-folder, named "Treasures" and "Sounds". Right click on the each of the new folders and use the Add -> Existing Item... function to add the files from the downloaded resources folders to corresponding folders in Visual Studio. Your Assets-folder should now look like this:



Modifying the XAML

Now we are going to modify the template's XAML so it has the basis of our UI. We are going to add a Pivot Control to the **MainPage.xaml** so that there's two pages (PivotItems): one for the treasure list, and one for the map. We are going to leave the treasure list page empty of XAML-elements, it will only have a `Grid` element as a root holder and rest of the UI content on that page is going to be generated dynamically in C#. In the map page, we are going to be adding the map XAML element to the page.

Note: The reason we aren't using the Pivot App -template is because this being a fairly simple application, the extra content created in the Pivot App -template, such as the ViewModel-folder, isn't necessary.

So, lets open **MainPage.xaml**.

First, you can delete all the XAML-code inside the `LayoutRoot` Grid-element tags, so the file looks like this:

```
<phone:PhoneApplicationPage
    x:Class="NFCTreasureHunt.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="Portrait" Orientation="Portrait"
    shell:SystemTray.IsVisible="False">

    <!--LayoutRoot is the root grid where all page content is placed-->
    <Grid x:Name="LayoutRoot" Background="DarkSlateGray">

    </Grid>

</phone:PhoneApplicationPage>
```

Next we can add the `Pivot` and `PivotItem` elements inside the `LayoutRoot` Grid. Let's give the two `PivotItems` headers "Treasures" and "Map" and the Pivot-element a title "NFC TREASURE HUNT". Also, let's place a Grid element inside both the `PivotItems` to hold the content of the two pages, and also give them names `TreasuresContentPanel` and `MapContentPanel`. Now your `LayoutRoot` grid element should look like this:

(C) Copyright Nokia 2013. All rights reserved.

```
<!--LayoutRoot is the root grid where all page content is placed-->
<Grid x:Name="LayoutRoot" Background="DarkSlateGray">
    <!--Pivot Control-->
    <phone:Pivot Title="NFC TREASURE HUNT">
        <!--Pivot item one-->
        <phone:PivotItem Header="Treasures">
            <Grid x:Name="TreasuresContentPanel" Grid.Row="1" Margin="12,0,12,0">

            </Grid>
        </phone:PivotItem>

        <!--Pivot item two-->
        <phone:PivotItem Header="Map">
            <Grid x:Name="MapContentPanel" Grid.Row="1" Margin="12,0,12,0">

            </Grid>
        </phone:PivotItem>
    </phone:Pivot>
</Grid>
```

As the last addition to the XAML, we are going to be adding a Map element to the map page. To do this first add this XAML code inside the `<phone:PhoneApplicationPage>` element in the beginning of the file. With this we add a reference to the Maps namespace that the XAML can use:

```
xmlns:maps="clr-namespace:Microsoft.Phone.Maps.Controls;assembly=Microsoft.Phone.Maps"
```

Finally, we add the Map element to the `MapContentPanel` content grid of the map `PivotItem`, by adding this code inside the Grid element:

```
<Grid x:Name="MapContentPanel" Grid.Row="1" Margin="12,0,12,0">
                <maps:Map x:Name="TreasureMap" Center="61.4984924, 23.7780349"
                            ZoomLevel="13"
                            CartographicMode="Hybrid"
                            Height="550"
                            VerticalAlignment="Top" />
</Grid>
```

You might have noticed some properties inside the Map element which are unique to the said element. Here's a short explanation about what each of the properties do:

- Center: Centers the map to given coordinates. This property replaces the Latitude and Longitude used in the old Maps API.
- ZoomLevel: Like the name suggests, this property controls on which level the map is zoomed to.
- CartographicMode: This property determines which kind of map is displayed, such as a plain terrain map, map with only roads visible or a hybrid of different options. Feel free to try out the different options by changing the property value with the other auto-fill options.

The Map class has various other unique properties to further customize and control your map element, you can find out more about them by taking a look at the Maps and Navigation for Windows Phone 8 at MSDN.

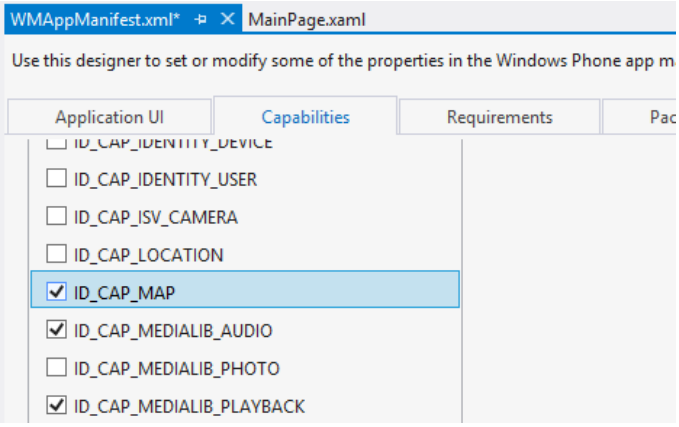
Finally, here is the complete **MainPage.xml** code:

```
<phone:PhoneApplicationPage
    x:Class="NFCTreasureHunt.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="Portrait" Orientation="Portrait"
    shell:SystemTray.IsVisible="False"
    xmlns:maps="clr-namespace:Microsoft.Phone.Maps.Controls;assembly=Microsoft.Phone.Maps">

    <!--LayoutRoot is the root grid where all page content is placed-->
    <Grid x:Name="LayoutRoot" Background="DarkSlateGray">
        <!--Pivot Control-->
        <phone:Pivot Title="NFC TREASURE HUNT">
            <!--Pivot item one-->
            <phone:PivotItem Header="Treasures">
                <Grid x:Name="TreasuresContentPanel" Grid.Row="1" Margin="12,0,12,0">

                </Grid>
            </phone:PivotItem>

            <!--Pivot item two-->
            <phone:PivotItem Header="Map">
                <Grid x:Name="MapContentPanel" Grid.Row="1" Margin="12,0,12,0">
                    <maps:Map x:Name="TreasureMap" Center="61.4984924, 23.7780349"
                                ZoomLevel="13"
                                CartographicMode="Hybrid"
                                Height="550"
                                VerticalAlignment="Top" />
                </Grid>
            </phone:PivotItem>
        </phone:Pivot>
    </Grid>

</phone:PhoneApplicationPage>
```

Adding Map Device Capabilities

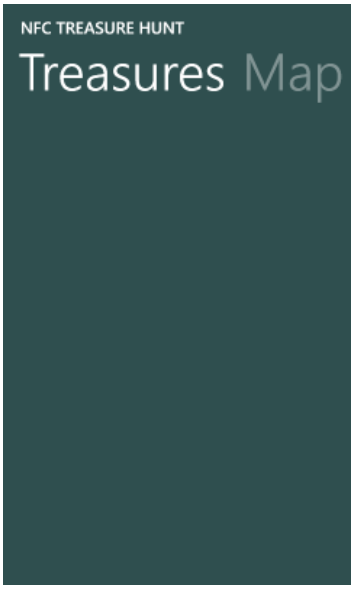
The last thing we need to do to is to add a capability support for map services. We do this by going to the **WMAppManifest.xml**-file which can be found under the Properties-folder in the Solution Explorer. In the file, go to the Capabilities-tab and check the box next to `ID_CAP_MAP`.

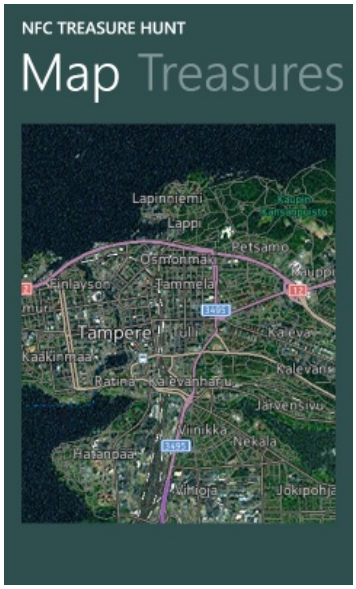
(C) Copyright Nokia 2013. All rights reserved.



You can now try running your application (you can still use the emulator at this point since there isn't any NFC functionality implemented yet). The application should have a Pivot view with the two pages "Treasures" and "Map". The former should be empty and the latter should show a map which is centred to Tampere, Finland.

Here's what the app should look like at this point:





Part 2: Adding NFC functionality, Creating and Filling the Treasure List

Next up, in this part we are going to be implementing the NFC functionality to the application, so that when the user holds his/hers WP8 device near to an NFC tag or a sticker, which has been written with a proper keyword, a treasure entry appears on a list on the "Treasures"-page. We also show a coin value on the list next to the treasure, so that different treasures are worth different amount of coins.

Creating a Dynamically Filling UI List With Grid Columns and Rows in C#

First, let's open the **MainPage.xaml.cs**-file and add some member declarations to the beginning of the class:

```
namespace NFCTreasureHunt
{
    public partial class MainPage : PhoneApplicationPage
    {
        Grid treasureGrid;
        ScrollViewer scrollView;
        Button clearTreasuresButton;
        int totalCoins;
        int treasureCount;
        TextBlock totalCoinsText;
        Image coinImage;
        MediaElement soundPlayer;

        // Constructor
        public MainPage()
        {
            ...
```

With these members, we place the `treasureGrid` inside the `ScrollView`, so that if the treasure list becomes so long that it stretches outside the screen, the `ScrollView` makes it possible for the user to scroll the list. The rest four members are pretty much self explanatory: the totalCoins keeps track of the total amount of coins accumulated from the collected treasures, `totalCoinsText` is the UI text element that displays the `totalCoins` and `clearTreasuresButton` is used to clear the treasures list. The `coinImage` is used to display an image of a coin next to the `totalCoinsText TextBox` and the soundPlayer is for playing sound effects. Finally, the `treasureCount` integer keeps track of how many treasures there are on the list.

Now we can add some code to the `MainPage()` constructor to initialize the class members we have declared and add some properties to them to place

(C) Copyright Nokia 2013. All rights reserved.

them in correct positions in the UI. We also add an event handler to the `clearTreasuresButton` and initialize the `soundPlayer` and add it to the `TreasureContentPanel`. So let's add some code to the constructor:

```
// Constructor
public MainPage()
{
    InitializeComponent();

    treasureCount = 0;
    totalCoins = 0;
    totalCoinsText = new TextBlock();

    clearTreasuresButton = new Button();

    soundPlayer = new MediaElement();
    this.TreasuresContentPanel.Children.Add(soundPlayer);

    scrollView = new ScrollViewer();
    scrollView.VerticalScrollBarVisibility = ScrollBarVisibility.Visible;
    Thickness t = new Thickness();
    t.Bottom = 150;
    scrollView.Padding = t;

    treasureGrid = new Grid();

    totalCoinsText.Text = totalCoins.ToString();
    totalCoinsText.VerticalAlignment = VerticalAlignment.Bottom;
    totalCoinsText.HorizontalAlignment = HorizontalAlignment.Right;
    totalCoinsText.Padding = new Thickness { Bottom = 75, Right = 75 };
    clearTreasuresButton.Content = "Clear treasures";
    clearTreasuresButton.VerticalAlignment = VerticalAlignment.Bottom;
    clearTreasuresButton.Background = new SolidColorBrush(Colors.Orange);
    clearTreasuresButton.Click += clearTreasuresButton_Click;

    scrollView.Content = treasureGrid;
    this.TreasuresContentPanel.Children.Add(scrollView);
    Grid totalCoinsGrid = new Grid();
    totalCoinsGrid.Children.Add(totalCoinsText);
    coinImage = new Image();
    coinImage.Source = new BitmapImage(new Uri("Assets/Treasures/coin.png", UriKind.Relative));
    coinImage.Width = 75;
    coinImage.Height = 75;
    coinImage.VerticalAlignment = VerticalAlignment.Bottom;
    coinImage.HorizontalAlignment = HorizontalAlignment.Right;
    coinImage.Margin = new Thickness { Bottom = 50 };
    totalCoinsGrid.Children.Add(coinImage);
    this.TreasuresContentPanel.Children.Add(totalCoinsGrid);
    this.TreasuresContentPanel.Children.Add(clearTreasuresButton);
}
```

Also, add the following namespaces to the beginning of the file to make the SolidColorBrush and BitmapImage -classes work:

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

Next up, we are going to create a `addTreasure(string)` method which is called when an nfc tag is read. It accepts a string as a parameter which is the keyword of which treasure the NFC tag or sticker holds. For example, if a tag would be written to hold a keyword "helmet", that keyword would be passed to the `addTreasure()` method which then adds a Helmet treasure entry to the treasure list.

The method is going to be quite many lines of code in length. First, we have a `TextBlock treasureEntry` to display the treasure's name which is gotten from the string `treasureName`, an Image treasureImage to display an image of the corresponding treasure on the left side of its name text, an integer that holds the treasure's coin worth and a Uri treasureImageUri that holds the path of the treasure's imagefile. We also add four ColumnDefinitions to the treasureGrid so that it pans correctly accross the width of the screen with its contents. After this we have conditions that check the given keyword and then sets the treasureName, coinCount and treasureImageUri accordingly, if the method received a keyword that didn't match any of the options, we set the properKeyword to false and then play an error sound. Otherwise we play a success sound. After the conditions we set the previously mentioned UI elements' properties so that they appear correctly on the same line in the UI. Finally we update the treasureCount and add the added treasure's coinCount to the totalCoins count and update the UI text.

Here is the complete code for the `addTreasure()`-method:

```
        void addTreasure(String treasure)
        {
            bool properKeyword = true;
            TextBlock treasureEntry;
            Image treasureImage;
            int coinCount = 0;
            String treasureName = "";
            Uri treasureImageUri = new Uri("Assets/Treasures/backpack.png", UriKind.Relative);
            treasureGrid.ColumnDefinitions.Add(new ColumnDefinition() { Width = GridLength.Auto });
            treasureGrid.ColumnDefinitions.Add(new ColumnDefinition() { Width = GridLength.Auto });
            treasureGrid.ColumnDefinitions.Add(new ColumnDefinition() { Width = GridLength.Auto });
            treasureGrid.ColumnDefinitions.Add(new ColumnDefinition() { Width = GridLength.Auto });
            treasureGrid.RowDefinitions.Add(new RowDefinition() { Height = GridLength.Auto });

            if (treasure == "backpack")
            {
                treasureImageUri = new Uri("Assets/Treasures/backpack.png", UriKind.Relative);
                coinCount = 30;
                treasureName = "Backpack";
            }
            else if (treasure == "axe")
            {
                treasureImageUri = new Uri("Assets/Treasures/axe.png", UriKind.Relative);
                coinCount = 15;
                treasureName = "Axe";
            }
            else if (treasure == "armor")
            {
                treasureImageUri = new Uri("Assets/Treasures/armor.png", UriKind.Relative);
                coinCount = 40;
                treasureName = "Armor";
            }
            else if (treasure == "coins")
            {
```


(C) Copyright Nokia 2013. All rights reserved.

```
            treasureImageUri = new Uri("Assets/Treasures/coins.png", UriKind.Relative);
            coinCount = 10;
            treasureName = "Stack of Coins";
        }
        else if (treasure == "dagger")
        {
            treasureImageUri = new Uri("Assets/Treasures/dagger.png", UriKind.Relative);
            coinCount = 8;
            treasureName = "Dagger";
        }
        else if (treasure == "helmet")
        {
            treasureImageUri = new Uri("Assets/Treasures/helmet.png", UriKind.Relative);
            coinCount = 13;
            treasureName = "Helmet";
        }
        else if (treasure == "potion_red")
        {
            treasureImageUri = new Uri("Assets/Treasures/potion.png", UriKind.Relative);
            coinCount = 7;
            treasureName = "Red Potion";
        }
        else if (treasure == "potion_blue")
        {
            treasureImageUri = new Uri("Assets/Treasures/potion_blu.png", UriKind.Relative);
            coinCount = 9;
            treasureName = "Blue Potion";
        }
        else if (treasure == "potion_green")
        {
            treasureImageUri = new Uri("Assets/Treasures/potion_green.png", UriKind.Relative);
            coinCount = 11;
            treasureName = "Green Potion";
        }
        else if (treasure == "shield")
        {
            treasureImageUri = new Uri("Assets/Treasures/shield.png", UriKind.Relative);
            coinCount = 25;
            treasureName = "Shield";
        }
        else if (treasure == "sword")
        {
            treasureImageUri = new Uri("Assets/Treasures/sword.png", UriKind.Relative);
            coinCount = 22;
            treasureName = "Sword";
        }
        else if (treasure == "tome")
        {
            treasureImageUri = new Uri("Assets/Treasures/tome.png", UriKind.Relative);
            coinCount = 18;
            treasureName = "Old Tome";
        }
        else if (treasure == "wand")
        {
            treasureImageUri = new Uri("Assets/Treasures/wand.png", UriKind.Relative);
            coinCount = 35;
            treasureName = "Magic Wand";
        }
        else
        {
            properKeyword = false;
        }

        // Play sound effects according to proper keyword received
        if (properKeyword)
        {
            // play succeed sound
            soundPlayer.Source = new Uri("Assets/Sounds/treasure_found.mp3", UriKind.Relative);
            soundPlayer.Play();
        }
        else
        {
            // play fail sound
            soundPlayer.Source = new Uri("Assets/Sounds/error.mp3", UriKind.Relative);
            soundPlayer.Play();
        }


        // Add treasure image and set name and coin count according to treasure
        treasureImage = new Image();
        treasureImage.Source = new BitmapImage(treasureImageUri);
        treasureImage.Width = 75;
        treasureImage.Height = 75;
        treasureImage.SetValue(Grid.ColumnProperty, 0);
        treasureImage.SetValue(Grid.RowProperty, treasureCount);
        treasureGrid.Children.Add(treasureImage);

        // Add treasure name TextBlock
        treasureEntry = new TextBlock();
        treasureEntry.Text = treasureName;
        treasureEntry.VerticalAlignment = VerticalAlignment.Center;
        treasureEntry.SetValue(Grid.ColumnProperty, 1);
        treasureEntry.SetValue(Grid.RowProperty, treasureCount);
        treasureGrid.Children.Add(treasureEntry);

        // Add coin count TextBlock
        treasureEntry = new TextBlock();
        treasureEntry.Text = coinCount.ToString();
        treasureEntry.VerticalAlignment = VerticalAlignment.Center;
        treasureEntry.Padding = new Thickness() { Left = 100.0 };
        treasureEntry.SetValue(Grid.ColumnProperty, 2);
        treasureEntry.SetValue(Grid.RowProperty, treasureCount);
        treasureGrid.Children.Add(treasureEntry);

        // Add coin image after coin count
        treasureImage = new Image();
        treasureImage.Source = new BitmapImage(new Uri("Assets/Treasures/coin.png", UriKind.Relative));
        treasureImage.HorizontalAlignment = HorizontalAlignment.Right;
        treasureImage.Width = 75;
        treasureImage.Height = 75;
        treasureImage.SetValue(Grid.ColumnProperty, 3);
        treasureImage.SetValue(Grid.RowProperty, treasureCount);
        treasureGrid.Children.Add(treasureImage);

        // Finally, we add the coin count of the treasure to the total coin count and increase the treasureCount by one
        // and update the totalCoinsText
        totalCoins += coinCount;
        totalCoinsText.Text = totalCoins.ToString();
        treasureCount++;
    }
```


(C) Copyright Nokia 2013. All rights reserved.

We now have the UI elements built to add treasures from an NFC read event, which we are going to implement next.

Creating the Global and NFCTreasure Classes, Implementing NFC Message Receiving

Before we implement the NFC message receiving functionality, we have to make some additions to the project:

First let's create a new class by right clicking the project in the Solution Explorer, selecting Add -> New Item... and select Code -> Class -type in the pop-up window. Name the class file **NFCTreasure.cs**. This is going to be a simple class with two string members: other for the keyword received from an NFC sticker and the other for storing the ID of the sticker. This ID is used to check that no more than one treasure can be added from one sticker. Here's what the `NFCTreasure`-class looks like:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Runtime.Serialization;

namespace NFCTreasureHunt
{
    public class NFCTreasure
    {
        public string type { get; set; }
        public string id { get; set; }

        public NFCTreasure(string type, string id)
        {
            this.type = type;
            this.id = id;
        }
    }
}
```

Next, let's create another class and name it `Global`. This class is going to be a static class. It's going to be used later on when we implement code for saving and resuming the state of the application when it's closed and opened again. For now we are going to add a member declaration of a List storing the `NFCTreasure` objects and an integer for storing the `coinCount` from the treasure list. So here's what the **Global.cs** looks like:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace NFCTreasureHunt
{
    static class Global
    {
        public static List<NFCTreasure> treasures { get; set; }
        public static int coinCount = 0;

        static Global()
        {
            treasures = new List<NFCTreasure>();
        }

    }
}
```

Now that we have the two classes created, we can finally implement the NFC message listening. Let's go back to **MainPage.xaml.cs** and add a new member declaration of `ProximityDevice` device to the beginning of the class. We also need to add a namespace Windows.Networking.Proximity:

```
...
using Windows.Networking.Proximity;

...

public partial class MainPage : PhoneApplicationPage
{
    Grid treasureGrid;
    ScrollViewer scrollView;
    Button clearTreasuresButton;
    int totalCoins;
    int treasureCount;
    TextBlock totalCoinsText;
    Image coinImage;
    MediaElement soundPlayer;
    ProximityDevice nfcDevice;

    ....
```

Next, in the constructor, we initialize the `ProximityDevice` object by setting it as a default NFC device the WP8 device holds. After that we add an event handler for the receiving of a message:

```
...
        // Constructor
        public MainPage()
        {
            InitializeComponent();

            nfcDevice = ProximityDevice.GetDefault();
            if (nfcDevice != null)
            {
                //Subscribe for plain text type NFC message
                long msgIf = nfcDevice.SubscribeForMessage("Windows.treasureKeyword", messageReceived);
            }

            ...
```


(C) Copyright Nokia 2013. All rights reserved.

With the `nfcDevice.SubscribeForMessage` -method we tell the NFC device to start listening for messages that are of the "Windows.treasure" message type. This is a custom created NFC message type. This means that when writing the treasure NFC tags with the keywords they have to be written with the same type or else the application ignores them when trying to read them. More about writing the NFC tags for this application, you can check out the "Extras"-chapter in the end of this article.

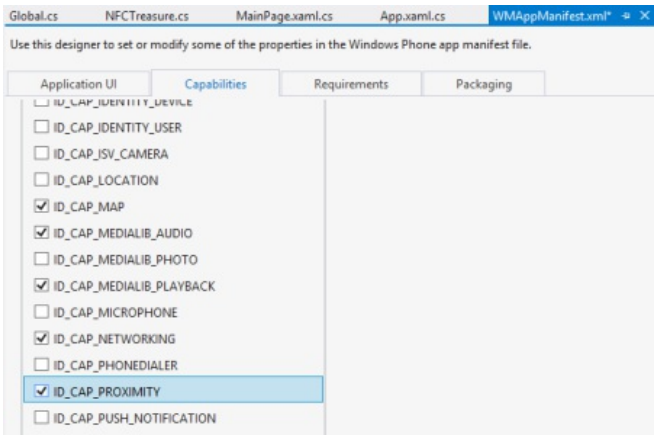
Next, we create the event handling method `messageReceived()`. It will take two parameters: `ProximityDevice sender`, which is the device sending the message (in our case the NFC tag/sticker) and `ProximityMessage message`, which holds the message we received (in this case just the treasure keyword). Inside the method, we make a check from the treasures List in Global that a tag or sticker with this ID hasn't been read before to stop duplication. If the check passes, we call the `addTreasure()` method created earlier with the keyword parsed to string from the message data. We also add a new NFCTreasure object to the List in the static Global class constructed from the NFC keyword and ID:

```
private void messageReceived(ProximityDevice sender, ProximityMessage message)
        {
            // Check if the treasure from this tag has already been added
            bool idFound = false;
            foreach (NFCTreasure treasure in Global.treasures)
            {
                if (treasure.id == sender.DeviceId)
                {
                    idFound = true;
                    break;
                }
            }

            // If this tag hasn't been read previously, we add the treasure
            if (!idFound)
            {
                addTreasure(message.DataAsString);
                Global.treasures.Add(new NFCTreasure(message.DataAsString, sender.DeviceId));
            }
        }
```

After this, we can create the `clearTreasuresButton_Click()` method which is called when the `clearTreasuresButton` is clicked. It is fairly simple: first we show a message box that ask for confirmation on if the user really wants to clear the list. If yes is answered, we clear the `treasureGrid`, set `totalCoinsText` to 0, clear the treasures List in Global and set the `treasureCount` and `Global.coinCount` to 0. `clearTreasuresButton_Click()`:

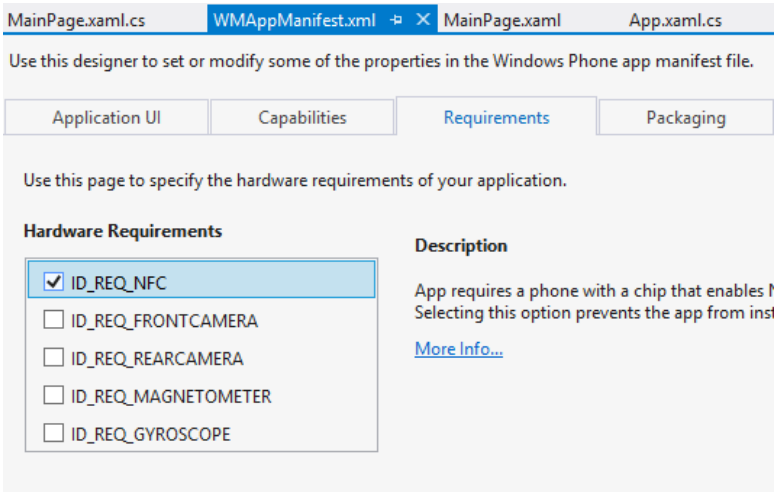
```
void clearTreasuresButton_Click(Object sender, EventArgs args)
        {
            // Ask for confirmation before clearing treasure list
            if (MessageBox.Show("Are you sure?", "Clear Treasures", MessageBoxButton.OKCancel) == MessageBoxResult.OK)
            {
                treasureGrid.Children.Clear();
                totalCoinsText.Text = "0";
                Global.treasures.Clear();
                treasureCount = 0;
                Global.coinCount = 0;
                totalCoins = 0;
            }
        }
```

Finally, to make NFC work on our application, we need to go to **WMAppManifest.xml**'s Capabilities tab and check the boxes next to `ID_CAP_PROXIMITY` and `ID_CAP_NETWORKING`:

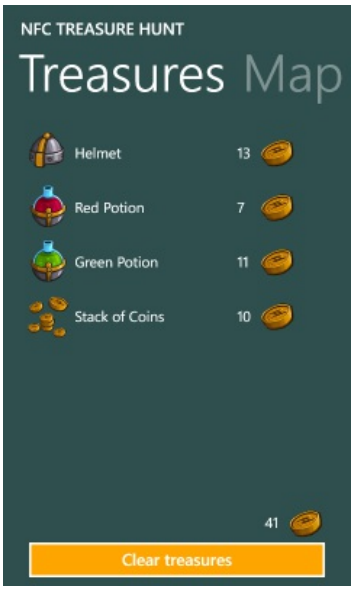


You can also go to the "Requirements"-tab on in the same file, and check the box next to `ID_REQ_NFC`. This forces the application to be not installed on non-NFC supported devices. You may also choose to not to add this requirement. Then the application can be installed on non-NFC devices, but obviously the NFC functionalities won't then work.

(C) Copyright Nokia 2013. All rights reserved.



Our application should now be ready to read NFC messages. You can try testing the app on your device, by holding the device on top of an NFC sticker that has been written with a proper keyword (check out the "Extras"-chapter at the end of this article for information on how to write proper NFC tags for this application), for example "helmet". You should hear the success sound effect and a new treasure entry should appear on the list. If the keyword isn't any of the expected ones, you'll hear the error sound and no entry has been added to the list. You can also test the "Clear Treasures" button and empty the list. Here's what the app looks like after receiving a few correctly keyworded messages from NFC stickers:



Part 3: Treasure Map, Getting Coordinates from Online Resource with WebClient and Adding them to the Map as Markers

In this part we are going to be adding functionality to the Map-page by filling the map with markings of treasure locations. We are going to get these locations' coordinates from an online resource using the `WebClient` class. We are also going to implement an option for the user to specify a web location of his/her own coordinates.

First, let's add two more class members to the **MainPage.xaml.cs**: a TextBox `coordinateUriTextBox` which holds the Uri for the online coordinate file and is editable by the user, and a Button `updateCoordinatesButton` that draws new map markers on the map element when new coordinate source `Uri` is given to the `coordinateUriTextBox`:

```
public partial class MainPage : PhoneApplicationPage
{
    Grid treasureGrid;
    ScrollViewer scrollView;
    Button clearTreasuresButton;
    int totalCoins;
    int treasureCount;
    TextBlock totalCoinsText;
    Image coinImage;
    ProximityDevice nfcDevice;
    MediaElement soundPlayer;

    TextBox coordinateUriTextBox;
    Button updateCoordinatesButton;

    // Constructor
    public MainPage()
    {
...
```

After this we initialize the two new members in the constructor and set them to the correct position in the UI by creating a Grid called `coordinateInfoGrid` for them and placing them inside it. We also give a default Uri for the `coordinatesUriTextBox`, which is a download link to a publicly accessible coordinate file I've uploaded to Dropbox for the purpose of this example:

(C) Copyright Nokia 2013. All rights reserved.

```
public MainPage()
{
    ...

    coordinateUriTextBox = new TextBox();
    Grid coordinateInfoGrid = new Grid();
    coordinateInfoGrid.VerticalAlignment = VerticalAlignment.Bottom;
    coordinateUriTextBox.Text = "https://dl.dropbox.com/s/us31jxwf2s4uxjp/coordinates.txt?dl=1";
    coordinateUriTextBox.Width = 300;
    coordinateUriTextBox.HorizontalAlignment = HorizontalAlignment.Left;
    coordinateInfoGrid.Children.Add(coordinateUriTextBox);

    Button updateCoordinatesButton = new Button();
    updateCoordinatesButton.Content = "Update";
    updateCoordinatesButton.Width = 150;
    updateCoordinatesButton.HorizontalAlignment = HorizontalAlignment.Right;
    updateCoordinatesButton.Background = new SolidColorBrush(Colors.Orange);
    updateCoordinatesButton.Click += updateCoordinatesButton_Click;
    coordinateInfoGrid.Children.Add(updateCoordinatesButton);

    this.MapContentPanel.Children.Add(coordinateInfoGrid);
}
```

The online coordinate file is a simple .txt file with the [latitude],[longitude] coordinates given each on a single line. The file's contents look like this:

```
61.4970698,23.7624787
61.5022199,23.7547325
61.4964861,23.7492393
61.497121,23.7902235
61.5033972,23.7835501
61.5041651,23.7732075
```

Next we are going to create a `getCoordinates()` method. Here we create a new `WebClient` class object and tell it to download the coordinates from the online resource as string. We pass the text from the `coordinateUriTextBox` as a parameter to the download method as the download source `Uri`. We also create an event handler for when the coordinate string has been downloaded:

```
void getCoordinates()
{
    WebClient client = new WebClient();
    client.DownloadStringCompleted += new DownloadStringCompletedEventHandler(client_DownloadStringCompleted);

    try
    {
        client.DownloadStringAsync(new Uri(coordinateUriTextBox.Text));
    }
    catch (Exception ex)
    {
        System.Diagnostics.Debug.WriteLine("Could not download string!");
    }
}
```

Now we create the event handler method `client_DownloadStringCompleted()` which takes the sender and result as parameters. In the method, we first parse the coordinates from the result by using Regex and string parsing methods (for this, you need to add a namespace to System.Text.RegularExpressions) and then create a `MapLayer` to which we add red rectangles as markers with `MapOverlays`. Finally, we tell the xaml Map element to center to the first given coordinate.

```
void client_DownloadStringCompleted(object sender, DownloadStringCompletedEventArgs e)
{
    string data = "";
    try
    {
        data = e.Result;
    }
    catch (Exception ex)
    {
        System.Diagnostics.Debug.WriteLine(ex.Message);
    }

    // Splits the coordinates to have one "[longitude],[latitude]" line in one array cell
    string[] coordinates = Regex.Split(data, "\n");

    MapLayer layer = new MapLayer();
    bool mapCentered = false;
    foreach(string s in coordinates)
    {
        // Splits the longitude and latitude to different array cells - longitude at [0] and latitude at [1]
        string[] split = s.Split(',');

        Rectangle rect = new Rectangle();
        rect.Fill = new SolidColorBrush(Colors.Red);
        rect.Height = 15;
        rect.Width = 15;

        try
        {
            MapOverlay overlay = new MapOverlay();
            overlay.Content = rect;
            overlay.GeoCoordinate = new GeoCoordinate(double.Parse(split[0]), double.Parse(split[1]));
            overlay.PositionOrigin = new Point(0.5, 0.5);
            layer.Add(overlay);

            // Center the map to the first coordinate on the list
            if (!mapCentered)
            {
                this.TreasureMap.Center = new GeoCoordinate(double.Parse(split[0]), double.Parse(split[1]));
                this.TreasureMap.ZoomLevel = 13.0;
                mapCentered = true;
            }
        }
        catch (Exception ex)
        {

        }
    }

    this.TreasureMap.Layers.Add(layer);
```


(C) Copyright Nokia 2013. All rights reserved.

```
}
```

For the previous code to work, add these using statements to the beginning of the file:

```
using Microsoft.Phone.Maps.Controls;
using System.Windows.Shapes;
using System.Device.Location;
using System.Text.RegularExpressions;
```

Now, what we have left to do is to add functionality to the `updateCoordinatesButton`. Lets create the event handler method `updateCoordinatesButton_clicked()`. Here we just clear any old map markers and call the `getCoordinates()`-method:

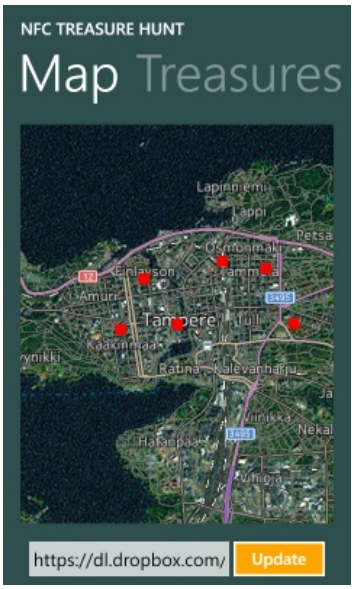
```
private void updateCoordinatesButton_Click(object sender, RoutedEventArgs e)
{
    // Clear the old markers from the map
    this.TreasureMap.Layers.Clear();
    getCoordinates();
}
```

And finally, let's add a call to the `getCoordinates()`-method to the bottom of the constructor, so the coordinates get loaded at the startup of the application:

```
public MainPage()
{
    ...

    ...

    getCoordinates();
}
```

Go ahead and run the application again. When you go to the "Map"-page, you should now see red dots on the map. If you want, you can try uploading your own coordinate file to some online file service where it's publicly available for download (or your own server) and paste the `Uri` to the text box and press the update button. If the file format is correct then new markers should appear on the coordinates you put in the file.

Here's how the "Map" page should look like now:



Part 4: Saving the Application State with IsolatedStorage

As a final part of this Example, we are going to implement code that saves the state of the application when it's closed or deactivated. For saving we are going to use the old IsolatedStorage API that has been in use since Windows Phone 7.

First, right click on the project in the Solution Explorer and choose Add -> New Item... and select Class-file from the Code category. Name the file **SaveGameData.cs**. Open the file and go ahead and delete the class code from inside the namespace. In place of that, we are going to create a public struct called `SaveGameData`. The struct is going to have three members: a List<NFCTreasure> where we put the treasures List from the static `Global` class, an integer to which we put the `coinCount` from the same class and a string where we store the `Uri` of the online map coordinates. These three things are the information we want saved when the application is deactivated or closed and loaded when the app is resumed or launched.

Our **SaveGameData.cs** should look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace NFCTreasureHunt
{
    public struct SaveGameData
    {
        public List<NFCTreasure> treasures;
```


(C) Copyright Nokia 2013. All rights reserved.

```
        public int coinCount;
        public string coordinateUri;
    }
}
```

Next, lets add one more class member to our Global class: a string coordinateUri, where we store the `coordinateUri` like in `SaveGameData` and give the dropbox coordinate file `Uri` we used in part 3 as the default value. We also need to add a `List<Type> knownTypesList` to the `Global` class. This list is used by `IsolatedStorage` to familiarize itself with developer created classes so objects of those type can be stored with it. In the constructor of the Global class we add our `NFCTreasure` class's type to the list:

```
static class Global
{
    public static List<NFCTreasure> treasures { get; set; }
    public static int coinCount = 0;
    public static List<Type> knownTypesList;
    public static string coordinateUri = "https://dl.dropbox.com/s/us31jxwf2s4uxjp/coordinates.txt?dl=1";

    static Global()
    {
        treasures = new List<NFCTreasure>();
        knownTypesList = new List<Type>();
        knownTypesList.Add(typeof(NFCTreasure));
    }

}
```

Now in the constructor in **MainPage.xaml.cs**, we need to change one line of code, so that the `coordinateUriTextBox` text is loaded from the `Global.coordinateUri` string:

```
// Constructor
public MainPage()
{
    ...

    coordinateUriTextBox.Text = Global.coordinateUri;

    ...
}
```

There we also make a new method called loadTreasureList() that goes through the Global.treasures List and adds treasures to the treasure list from it:

```
void loadTreasureList()
        {
            foreach (NFCTreasure treasure in Global.treasures)
            {
                addTreasure(treasure.type);
            }
        }
```

We also add a call to the `loadTreasureList()` method to the constructor, so that it's called when the application launches or is resumed. This way, the saved treasures get loaded to the UI treasure list:

```
// Constructor
public MainPage()
{
    ...

    ...

    loadTreasureList();
}
```

Next, we need to add some things to the `NFCTreasure` class code to make it storeable with `IsolatedStorage`. First we add a `[DataContract]` tag on top of the class name and after that two `[DataMember]` tags on top of the two class members. We also need to add a using statement to System.Runtime.Serialization. Our **NFCTreasure.cs** should then look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Runtime.Serialization;

namespace NFCTreasureHunt
{
    [DataContract]
    public class NFCTreasure
    {
        [DataMember]
        public string type { get; set; }
        [DataMember]
        public string id { get; set; }

        public NFCTreasure(string type, string id)
        {
            this.type = type;
            this.id = id;
        }
    }
}
```

Now, let's open up **App.xaml.cs** which can be found under the **App.xaml**-file in the Solution Explorer. First add the name space beginning of the file:

```
using System.IO.IsolatedStorage;
using System.Runtime.Serialization;
```

Next go ahead and find the `Application_Deactivated()`-method. Here we are going to put the `IsolatedStorage` code that saves the `SaveGameData`

(C) Copyright Nokia 2013. All rights reserved.

struck to the application's own specific isolated storage (which we name "NFCTreasureHunt_Save"). In the beginning of the method we create a new `SaveGameData` struct object and bring the information from our `Global` class to it. After this we use `IsolatedStorageFile` and `IsolatedStorageFileStream` to write the struct to the application storage using a `DaraContractSerializer` object (for which we give the `Global.knownTypesList` as a parameter). Here is the code for the method:

```
// Code to execute when the application is deactivated (sent to background)
// This code will not execute when the application is closing
private void Application_Deactivated(object sender, DeactivatedEventArgs e)
{
    SaveGameData data = new SaveGameData();
    data.treasures = Global.treasures;
    data.coinCount = Global.coinCount;
    data.coordinateUri = Global.coordinateUri;

    using (IsolatedStorageFile storage = IsolatedStorageFile.GetUserStoreForApplication())
    {
        using (IsolatedStorageFileStream fileStream = new IsolatedStorageFileStream("NFCTreasureHunt_Save", System.IO.FileMode.Creat
        {
            DataContractSerializer serializer = new DataContractSerializer(typeof(SaveGameData), Global.knownTypesList);
            serializer.WriteObject(fileStream, data);
            fileStream.Close();
        }
    }

    System.Diagnostics.Debug.WriteLine("Saving");
}
```

Now go ahead and copy the code inside the `Application_Deactivated()`-method to the `Application_Closing()` method. We do this so that the application state gets saved whether the application is closing or just being deactivated:

```
// Code to execute when the application is closing (eg, user hit Back)
// This code will not execute when the application is deactivated
private void Application_Closing(object sender, ClosingEventArgs e)
{
    SaveGameData data = new SaveGameData();
    data.treasures = Global.treasures;
    data.coinCount = Global.coinCount;
    data.coordinateUri = Global.coordinateUri;

    using (IsolatedStorageFile storage = IsolatedStorageFile.GetUserStoreForApplication())
    {
        using (IsolatedStorageFileStream fileStream = new IsolatedStorageFileStream("NFCTreasureHunt_Save", System.IO.FileMode.Creat
        {
            DataContractSerializer serializer = new DataContractSerializer(typeof(SaveGameData), Global.knownTypesList);
            serializer.WriteObject(fileStream, data);
            fileStream.Close();
        }
    }

    System.Diagnostics.Debug.WriteLine("Saving");
}
```

Now find the method `Application_Activated()` from the same file. Here we are going to add the code to read the application's isolated storage data. The code does pretty much the opposite of what the saving code does: it creates a `SaveGameData` struct object and then deserializes the storage data into that object. The values from the object then get copied into the `Global` class's members:

```
// Code to execute when the application is activated (brought to foreground)
// This code will not execute when the application is first launched
private void Application_Activated(object sender, ActivatedEventArgs e)
{
    using (IsolatedStorageFile storage = IsolatedStorageFile.GetUserStoreForApplication())
    {
        if (storage.FileExists("NFCTreasureHunt_Save"))
        {

            System.Diagnostics.Debug.WriteLine("Loading");

            SaveGameData data = new SaveGameData();

            using (IsolatedStorageFileStream fileStream = new IsolatedStorageFileStream("NFCTreasureHunt_Save", System.IO.FileMode.O
            {
                DataContractSerializer serializer = new DataContractSerializer(typeof(SaveGameData), Global.knownTypesList);
                data = (SaveGameData)serializer.ReadObject(fileStream);
            }

            Global.coinCount = data.coinCount;
            Global.treasures = data.treasures;
            Global.coordinateUri = data.coordinateUri;
            System.Diagnostics.Debug.WriteLine(data.coordinateUri);
        }
        else
        {

        }
    }
}
```

Also add the same code to the `Application_Launching()`-method:

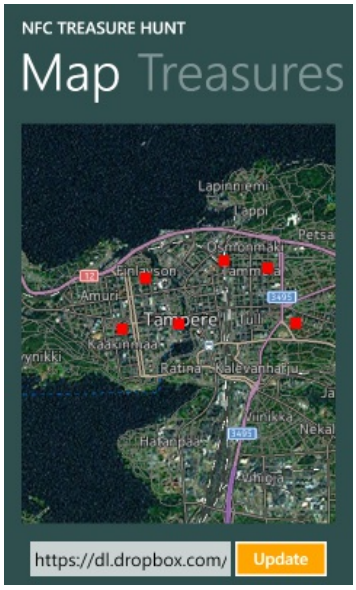
```
// Code to execute when the application is launching (eg, from Start)
// This code will not execute when the application is reactivated
private void Application_Launching(object sender, LaunchingEventArgs e)
{
    using (IsolatedStorageFile storage = IsolatedStorageFile.GetUserStoreForApplication())
    {
        if (storage.FileExists("NFCTreasureHunt_Save"))
        {

            System.Diagnostics.Debug.WriteLine("Loading");
```


(C) Copyright Nokia 2013. All rights reserved.

```
            SaveGameData data = new SaveGameData();

            using (IsolatedStorageFileStream fileStream = new IsolatedStorageFileStream("NFCTreasureHunt_Save", System.IO.FileMode.O
            {
                DataContractSerializer serializer = new DataContractSerializer(typeof(SaveGameData), Global.knownTypesList);
                data = (SaveGameData)serializer.ReadObject(fileStream);
            }

            Global.coinCount = data.coinCount;
            Global.treasures = data.treasures;
            Global.coordinateUri = data.coordinateUri;
            System.Diagnostics.Debug.WriteLine(data.coordinateUri);
        }
        else
        {

        }
    }
}
```

And there we go! That should be all the code required to save the application state. You can once again go ahead and run the application. Now, you can try for example adding a few treasures to the list and changing the coordinate Uri on the "Map"-page, then deactivating the application by pressing the start button on your WP8 device, and then starting the application again. The treasures you had in the list should still be there and the coordinate Uri you gave to the text box on the map page should also be the same instead of the default one.

The treasure hunting application is now finished! Here's what the final version should look like:



If you still want to know more about writing the treasure hunting NFC tags and showing your own location on the treasure map using GPS, please take a look at the following "Extras" chapter. Otherwise, I hope you found this example article useful!

Extras

Writing NFC Treasure Tags

As you probably noticed, the article hasn't so far told you how to write the treasure tags. I wanted to create a separate chapter for it, because in my opinion the writer application should be separate to the hunting application.

In this chapter I will focus on explaining the NFC writing part of the application and not go into detail on creating the UI or other parts of it since those parts will be simple.

First, let's create another normal Windows Phone App project in Visual Studio. Name the project "NFCTreasureHuntTagWriter". Like with the main application, target this for Windows Phone OS 8.0.

First, here is the code for **MainPage.xaml**:

```
<phone:PhoneApplicationPage
    x:Class="NFCTreasureHuntTagWriter.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="Portrait" Orientation="Portrait"
    shell:SystemTray.IsVisible="False">

    <!--LayoutRoot is the root grid where all page content is placed-->
    <Grid x:Name="LayoutRoot" Background="DarkSlateGray">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
```

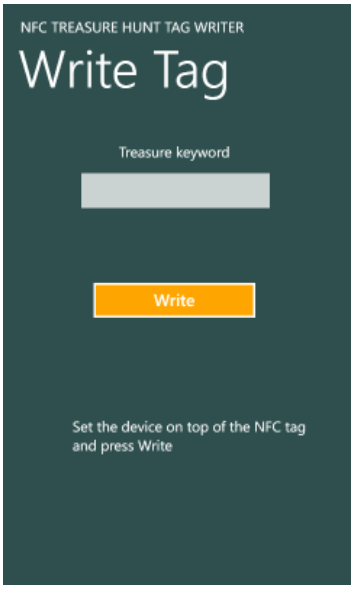

(C) Copyright Nokia 2013. All rights reserved.

```
        <!--TitlePanel contains the name of the application and page title-->
        <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
            <TextBlock Text="NFC TREASURE HUNT TAG WRITER" Style="{StaticResource PhoneTextNormalStyle}" Margin="12,0"/>
            <TextBlock Text="Write Tag" Margin="9,-7,0,0" Style="{StaticResource PhoneTextTitle1Style}"/>
        </StackPanel>

        <!--ContentPanel - place additional content here-->
        <Grid x:Name="ContentPanel" Grid.Row="1" Margin="14,0,10,0">
            <TextBlock HorizontalAlignment="Left" TextWrapping="Wrap" VerticalAlignment="Top" RenderTransformOrigin="0.529,-0.111" M
            <TextBox HorizontalAlignment="Left" Height="72" Margin="82,60,0,0" TextWrapping="Wrap" Text="" Name="keywordTextBox" Ver
            <Button Content="Write" Click="Button_Click_1" HorizontalAlignment="Left" Margin="99,211,0,0" VerticalAlignment="Top" Wi
            <TextBlock Name="infoText" Text="Set the device on top of the NFC tag and press Write" HorizontalAlignment="Left" Margin
            </TextBlock>

        </Grid>
</Grid>

</phone:PhoneApplicationPage>
```

And here's the code for **MainPage.xaml.cs**:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Navigation;
using Microsoft.Phone.Controls;
using Microsoft.Phone.Shell;
using NFCTreasureHuntTagWriter.Resources;
using Windows.Networking.Proximity;
using Windows.Storage.Streams;

namespace NFCTreasureHuntTagWriter
{
    public partial class MainPage : PhoneApplicationPage
    {
        ProximityDevice nfcDevice;

        // Constructor
        public MainPage()
        {
            InitializeComponent();

            nfcDevice = ProximityDevice.GetDefault();
        }

        private void Button_Click_1(object sender, RoutedEventArgs e)
        {
            DataWriter dataWriter = new DataWriter() { UnicodeEncoding = UnicodeEncoding.Utf8 };
            dataWriter.WriteString(this.keywordTextBox.Text);

            if (nfcDevice != null)
            {
                nfcDevice.PublishBinaryMessage("Windows:WriteTag.treasureKeyword", dataWriter.DetachBuffer(), messageSent);
            }
            else
            {
                this.infoText.Text = "NFC not supported!";
            }
        }

        private void messageSent(ProximityDevice sender, long messageId)
        {
            //After tag is written, inform the user and stop publishing the message
            nfcDevice.StopPublishingMessage(messageId);
            Message.Show("Tag succesfully written!");
        }
    }
}
```

And here's what the application looks like:



What happens in the application is that when the write button gets clicked, it tells the `ProximityDevice nfcDevice` to start publishing the keyword given to the textfield converted to a binary message with the `DataWriter` class. We use the binary publishing method, because NFC text is required to be in

(C) Copyright Nokia 2013. All rights reserved.

UTF8-encoding and with the `DataWriter` we can make sure it is. This message is published with the custom protocol "Windows.treasureKeyword" which we used in the hunter application to receive the NFC message. This time, when we are writing to a tag, we add the word "WriteTag" to the protocol so it becomes "Windows:WriteTag.treasureKeyword".

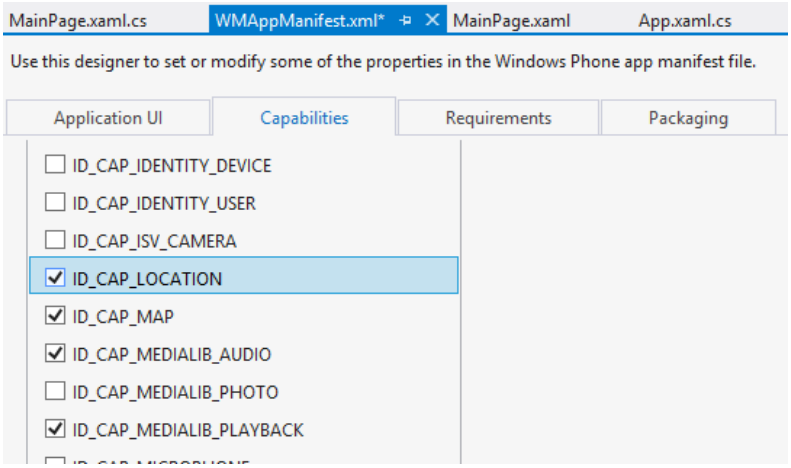
When the `nfcDevice` is successful in writing the message, it calls the `messageSent()` method where we simply stop publishing the message and inform the application user via a `MessageBox` that the tag writing was a success.

Showing Own Location on the Treasure Map

In my opinion, a map showing your own location, isn't a treasure map :). However, it is still a good idea to show how it's done since the subject is very close to the other map functions this article includes and because the Windows Phone Runtime Location API in Windows Phone 8 is new. We aren't going to go into the theory part of the new Location API in this article because you can find a good documentation about it at MSDN at Location for Windows Phone 8.

This is a short example on how to get the device's location continuously with GPS and add it as a marker on the treasure map. We use the NFCTreasureHunt project we previously created.

First enable the `ID_CAP_LOCATION` capability in the **WMAppManifest.xml**-file:



Now we move to **MainPage.xaml.cs**. First add a using statement for Windows.Devices.Geolocation; to the beginning of the file. After this, we create a new `Geolocator` class member and a new MapLayer to the beginning of the class declaration:

```
public partial class MainPage : PhoneApplicationPage
{
    ...

    Geolocator geolocator;
    MapLayer locationLayer;

    // Constructor
    public MainPage()
    {

    ...
```

After this, we initialize the geolocator in the constructor, set the `DesiredAccuracy` and `MovementTreshold` properties and set an event handler for what happens when the `geoposition` of the device changes. We also initialize the new`locationLayer` and add it as a layer of the `TreasureMap`:

```
// Constructor
        public MainPage()
        {
            InitializeComponent();

            geolocator = new Geolocator();
            geolocator.DesiredAccuracy = PositionAccuracy.High;
            geolocator.MovementThreshold = 10; // Meters
            geolocator.PositionChanged += geolocator_PositionChanged;

            locationLayer = new MapLayer();
            this.TreasureMap.Layers.Add(locationLayer);

            ...
```

Now the only thing left to do is to create the `geolocator_PositionChanged` event handling method. In the method we simply take the latitude and longitude coordinates of the device from the `PositionChangedEventArgs` we get as a parameter and create a new map marker on those coordinates:

```
private void geolocator_PositionChanged(Geolocator sender, PositionChangedEventArgs args)
{
    // Clear previous mark from the map
    locationLayer.Clear();

    Dispatcher.BeginInvoke(() =>
    {
        Rectangle rect = new Rectangle();
        rect.Fill = new SolidColorBrush(Colors.Blue);
        rect.Height = 15;
        rect.Width = 15;

        MapOverlay overlay = new MapOverlay();
        overlay.Content = rect;
```


(C) Copyright Nokia 2013. All rights reserved.

```
        overlay.GeoCoordinate = new GeoCoordinate(double.Parse(args.Position.Coordinate.Latitude.ToString()), double.Parse(args.Position.Coordinate.Longitude.ToS
        overlay.PositionOrigin = new Point(0.5, 0.5);
        locationLayer.Add(overlay);
    });
}
```

And there we go! Now, there should be a blue marker on the treasure map showing the device's position. When the device moves 10 or more meters, the `geolocator_PositionChanged` event handler method is called, and the map marker is drawn to a new position.

Features Documentation Resources

Maps and location:

- Maps and Navigation for Windows Phone 8
- Location for Windows Phone 8

Proximity/NFC:

- Proximity for Windows Phone 8

IsolatedStorage:

- Using IsolatedStorage on the Phone

Summary

In this article we showed you an example about how to implement the new NFC, Location, and Nokia Maps features in your Windows Phone 8 application.

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.