

# Creating a Real-World, GPS-enabled WP8 Tag Game

This article explains how to create a tag game in Windows Phone 8 with the help of newly supported technologies like Windows Azure, GPS, and NFC.

 **Note:** This is a somewhat large tutorial, so if you would just like to learn about the technologies in use, feel free to skip to the end where you'll find a "Recap" section.

 **Warning:** This app's location wrapper converts the current location to a string, but in most apps returning either a custom Location type or the default GeoCoordinate could be better.

## Creating a Real-World, GPS-Enabled WP8 Tag Game with NFC, the Location API, and the Nokia Map Control



In this tutorial, we will create a WP8 tag game that works by combining NFC, the Location API, and the new Nokia `<Maps />` Control. The game will facilitate the playing of a large-scale "hide and go seek"-like game across, for instance, a large city like San Francisco. The game will work as follows:

- The tagger will start our app and press "new game" to initiate a new game of tag.
- The tageses will run and use NFC to tap into the game.
- Every 10 meters that a tagee runs, the app will automatically update their position.
- The tagger will see these updates in a Nokia Map control and be able to run to them.
- When the tagger finds a tagee, they will tap phones and the tagee will become the tagger, and vice versa.

## Why do we use...

Throughout this article, we'll be using new technologies and services like Windows Azure, NFC, the Location API, and the new Nokia `<Maps />` Control, and there are reasons why we've chosen these over their alternatives.

- **Azure** - We're using Azure Mobile Services to store the user's checkins in the cloud. Mobile Services is simpler to use than something like a custom API because there are premade SDKs for .NET languages - we don't have to worry about the actual network calls and everything, all we have to do is perform operations on a table.
- **NFC** - We're using NFC to connect other users to the game. The alternatives would be to use Bluetooth (which uses too much battery, and wasn't designed for this type of "tapping" action) or an "enter this code" type solution (which is cumbersome for the user).
- **The Location API** - We're using the Location API to get the current user's location, and then update that with Windows Azure.
- **Nokia `<Maps />` Control** - We're using the Nokia `<Maps />` control to display the location of the tageses to the tagger.

## Notes

- Since I don't have an NFC capable WP8 device to test this on, I can't confirm whether this app works in the real-world. The code provided here is a "should-work" solution, but has not been tested. If you have an NFC capable WP8 device and would like to test this, please edit this tutorial with any code fixes you find.
- This is an app that was designed to provide a good overview of using Windows Azure, location data, NFC, and Nokia Maps together. Since it uses Windows Azure to handle checkins, it can potentially use a lot of data on the phone and space on Azure. A real-world solution would probably be better off using its own API.

## Before you Start

Create a new Windows Phone 8 project in Visual Studio called "GPSTag" to begin.

## Capabilities

In the Visual Studio solution explorer window, expand "Properties", double click **WMAppManifest.xml**, and then select the Capabilities tab. Make sure that the following capabilities are selected:

```
ID_CAP_LOCATION
ID_CAP_MAP
ID_CAP_MEDIALIB_AUDIO
```

```
ID_CAP_MEDIALIB_PLAYBACK
ID_CAP_NETWORKING
ID_CAP_PROXIMITY
ID_CAP_WEBBROWSERCOMPONENT
```

Switch to the Requirements tab and select "ID\_REQ\_NFC" to mark that a phone must have NFC to use this app.

## Setting up the Frontend

We will be using a few pages in our app, so here's how to set them up:

- In your **MainPage.xaml** page, drag two buttons onto the default grid and modify them to read "Host Game" and "Join Game."
- Create (Ctrl-Shift-A) a new Portrait Page and name it **StartGame.xaml**. Drag a button onto the grid and modify it to read "continue."
- Create a new Portrait Page and name it **JoinGame.xaml**.
- Create a new Portrait Page and name it **Tagger.xaml**. Copy and paste:

```
<maps:Map HorizontalAlignment="Left" Margin="10,10,0,0" VerticalAlignment="Top"
Height="599" Width="436" x:Name="Map" LandmarksEnabled="true"
PedestrianFeaturesEnabled="True" />
```

into the ContentPanel grid to create a new Nokia Maps control.

- Create a new Portrait Page and name it **Tagee.xaml**.

## Windows Azure

We'll be storing our check-ins with Windows Azure Mobile Services (a newly supported service available on WP8, Windows 8, and iOS that makes storing and retrieving data easy), so first you need to sign up for Azure. Visit [the Azure Free Trial sign up page](#) and sign up for the free 3-month trial. Once you've done that, it's time to set up the actual mobile service our app will use. When you visit the [Azure Management Portal](#) you should see a "NEW" button in the bottom right. Click that, and then choose "Mobile Service" under "Compute." Follow the wizard to set up a new mobile service (name it whatever you want) and then continue. After the mobile service is created, clicking on it should bring you to the dashboard. Expand "Connect an existing Windows Store app" and follow the instructions there to connect the app we made in the last section to Azure Mobile Services. Mobile Services work by creating tables which hold custom data types. We'll need to create a table to hold our checkins. To do that, go back to the Management Portal, click on the mobile service, then choose the "data" tab and add a new "Checkin" table.

## "Random File"

We'll need to have a few classes easily accessible to all parts of the application (such as our location wrapper, Checkin type, etc.) so create a new Class (Shift-Alt-C) and name it Types.cs (or anything that suits your fancy). Replace the default using statements with:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Windows.Devices.Geolocation;
using System.IO.IsolatedStorage;
using System.Windows;
```

The first thing we'll add is a "Checkin" type for our Mobile Service. Replace the default class in the file with the following:

```
public class Checkin
{
    public int Id { get; set; }
    public int playerId { get; set; }
    public string location { get; set; }
```

```

public string Comments { get; set; }
public int gameId { get; set; }
}

```

Basically, the above code defines a public "Checkin" class that contains data about the user's current location, playerId, and any extra comments we want to attach to the checkin. Note that `Id` is a required value that you should have in your Mobile Service types. It's automatically incremented by the Mobile Service SDK, so you don't need to do anything with it but have it there (in fact, if you attempt to modify it the SDK will throw an error).

Since our code is going to be using a lot of location data, it would be nice to have a simple wrapper that will handle location for us. Note that if you use location data in your app, you must make sure the user knows and gives permission for you to use that data. Our `locationWrapper` simplifies that by automating the confirmation dialog. Copy and paste the below right after the `Checkin` class:

```

public class locationWrapper
{
    //modified from http://msdn.microsoft.com/en-
    US/library/windowsphone/develop/jj206956(v=vs.105).aspx
    public static async Task<bool> GetConfirmation()
    {
        if (IsolatedStorageSettings.ApplicationSettings.Contains("doLocation"))
        {
            return
            Convert.ToBoolean(IsolatedStorageSettings.ApplicationSettings["doLocation"]);
        }
        else
        {
            MessageBoxResult result = MessageBox.Show("We're about to use your location.
            Are you OK with that?", "Location", MessageBoxButton.OKCancel);
            if (result == MessageBoxResult.OK)
            {
                IsolatedStorageSettings.ApplicationSettings["doLocation"] = true;
            }
            else
            {
                IsolatedStorageSettings.ApplicationSettings["doLocation"] = false;
            }
            IsolatedStorageSettings.ApplicationSettings.Save();
            return result == MessageBoxResult.OK;
        }
    }
    public static async Task<string> GetCurrentLocation()
    {
        if (await GetConfirmation())
        {
            Geolocator locator = new Geolocator();
            locator.DesiredAccuracyInMeters = 1; //we want this as accurate as possible
            //http://msdn.microsoft.com/en-
            US/library/windowsphone/develop/jj206956(v=vs.105).aspx
            try
            {
                Geoposition geoposition = await locator.GetGeopositionAsync(
                maximumAge: TimeSpan.FromMinutes(5),
                timeout: TimeSpan.FromSeconds(10)
                );
                return geoposition.Coordinate.Latitude.ToString("0.00") + ":" +

```

```

geoposition.Coordinate.Longitude.ToString("0.00");
    }
    catch (Exception ex)
    {
        if ((uint)ex.HResult == 0x80004004)
        {
            MessageBox.Show("Location seems to be turned off in the phone settings,
please turn it on.");
        }
        return "0:0";
    }
    }
else
{
    return "0:0";
}
}
}

```

The code above allows you to call `GetCurrentLocation()` which prompts the user to accept location collection (and caches that decision), then fetches the current location and returns it in the form of latitude:longitude. If an error occurs, it returns 0:0.

Finally, copy and paste the below:

```

public class BufferFuncs
{
    public static Windows.Storage.Streams.IBuffer GetBufferFromString(string str)
    {
        using (var dw = new Windows.Storage.Streams.DataWriter())
        {
            dw.UnicodeEncoding = Windows.Storage.Streams.UnicodeEncoding.Utf16LE;
            dw.WriteString(str);
            return dw.DetachBuffer();
        }
    }
}

```

We'll use this code to write a string's buffer to an NFC tag later.

## MainPage

The MainPage page simply acts as a gateway to let the user chose whether they want to host or join a game. Double-click the Host Game button in the Design View and add `App.RootFrame.Navigate(new Uri("/StartGame.xaml"));` to the newly created click handler. Do the same for the Join Game button, but instead use `App.RootFrame.Navigate(new Uri("/JoinGame.xaml"));`

## StartGame

The StartGame page transmits an NFC tag which allows other devices to connect with it and join the game.

## Using Statements

Open **StartGame.xaml.cs**, then add the following using statements to the top of the page:

```

using Windows.Networking.Proximity;
using Microsoft.WindowsAzure.MobileServices;
using System.Windows.Threading;

```

Windows.Networking.Proximity contains functions that let use NFC, Bluetooth, and other proximity devices, , Microsoft.WindowsAzure.MobileServices is required for Mobile Services to work, and we'll use System.Windows.Threading to create a timer.

## NFC

Now that we have the correct using statements, we need to add some functions and variables that we'll use later. Add the following inside of the StartGame class:

```
int currentPlayers = 0;
long messageID = -1;
ProximityDevice device = ProximityDevice.GetDefault();
IMobileServiceTable<Checkin> CheckinTable = App.MobileService.GetTable<Checkin>();
```

Here's an explanation of each part that we just added:

- currentPlayeres is a count of how many players we have so far. We'll use it to alert the user when a new player has joined.
- messageID is the ID of the NFC message we're transmitting. We'll use this to stop publishing our message once we start the game.
- device is a proximity device. If this is null, it means the device doesn't support NFC.
- checkinTable is our Azure Mobile Services table. We'll perform operations against this to update the table on Azure's datacenters.

Now that we have the variables that we'll need, copy the following code into the same file:

```
private async void StartNewGame()
{
    List<Checkin> playerIds = await CheckinTable.OrderByDescending(a =>
a.playerId).ToListAsync(); //get the highest current playerId
    App.playerId = playerIds[0].playerId + 1; //set our player id to one more than that

    List<Checkin> gameIds = await CheckinTable.OrderByDescending(a =>
a.gameId).ToListAsync(); //do the same for our game
    App.gameId = gameIds[0].gameId + 1; //set our game id to one more than that
    if (device != null)
    {
        messageID = device.PublishBinaryMessage("Windows.GPSTagGame",
BufferFuncs.GetBufferFromString(App.gameId.ToString()), MessageSent);
    }
    else
    {
        MessageBox.Show("Sorry, your phone doesn't support NFC.");
    }
    Checkin beginingCheckin = new Checkin();
    beginingCheckin.playerId = App.playerId;
    currentPlayers++;
    beginingCheckin.gameId = App.gameId;
    beginingCheckin.Comments = "Game Init";
    var loc = await locationWrapper.GetCurrentLocation();
    if (loc != "0:0")
    {
        beginingCheckin.location = loc;
        await CheckinTable.InsertAsync(beginingCheckin);
        MessageBox.Show("You're all set - tap your phone to other phones running GPSTag
to begin a game.");
    }
    else
```

```

    {
        MessageBox.Show("Something went wrong and we can't continue.");
    }
}

```

In the code above, we're trying to create a checkin with new game and player IDs, then insert it into the Azure Mobile Services Table. We first need to know what the current game and player IDs are and then go one above that to ensure that our game doesn't get confused with another one. To achieve that, we'll order the checkin table that we created above by playerId to get the highest player ID, and then we'll make our playerId one more than that. After we do the same for our gameId, we can get to the actual NFC.

If the device we're on right now doesn't have NFC support, then the ProximityDevice will be null. Using a simple if/then statement, we can check whether NFC is supported, and if it isn't we can go ahead and return an error now. If the phone does have NFC support, we use the `PublishBinaryMessage` function of our ProximityDevice to transmit the gameId to other phones running our app.

While the message is transmitting, we need to actually start the game. To do this, we create a new Checkin variable and fill it in with the default information. Once we have the checkin, we can insert it into our `IMobileServiceTable` using `InsertAsync`. Note that once you use `InsertAsync` there's nothing else you need to do to get Azure to upload and sync the checkin – the SDK handles it all automatically.

But what happens once a device connects? In the code above, you'll notice a reference to `MessageSent`, which is the handler that will be called when our message is successfully sent to another device, at which point the app should alert the user to there being a new player. To work around the fact that there isn't unlimited speed on the Internet, and it might take some time for the new player to checkin to the database, we're going to create a `DispatcherTimer` that's set to go off in 2 seconds which will call our actual user-checking code.

```

DispatcherTimer timr = new DispatcherTimer();
private void MessageSent(ProximityDevice sender, long messageid)
{
    timr.Interval = new TimeSpan(0, 0, 2); //give the other phone 2 seconds to join the
game
    timr.Tick += timr_Tick;
    timr.Start();
}
async void timr_Tick(object sender, EventArgs e)
{
    timr.Stop();
    var nowPlayers = (await App.MobileService.GetTable<Checkin>().Where(a => a.gameId ==
App.gameId).ToListAsync()).Count;
    if (nowPlayers != currentPlayers)
    {
        MessageBox.Show("A new player has joined the game! Current players: " +
nowPlayers);
    }
    currentPlayers = nowPlayers;
}

```

Once the timer goes off, we'll refresh the list of current players in our game, and if someone new joins (if `currentPlayers` is less than the number of checkins for our gameId in the checkin table) the user will be alerted.

Go back to the design view for **StartGame.xaml** (you can do this by double-clicking **StartGame.xaml** in the solution explorer), and double click the "continue" button to create a new click handler. Replace the default code with:

```

device.StopPublishingMessage(messageID);
if (currentPlayers > 1)
{

```

```

    App.RootFrame.Navigate(new Uri("/Tagger.xaml"));
}
else
{
    MessageBox.Show("You need at least two players, including yourself, to begin a
game");
}

```

`device.StopPublishingMessage` will stop publishing the NFC message we were publishing before (notice how we're using the `messageID` variable now). If there's more than one player, we'll send them to the Tagger page, but if not we'll instruct them to get more people playing.

## Tagger

### Using Statements

Open **Tagger.xaml.cs** (you can see the `.cs` file by expanding **Tagger.xaml** in the solution explorer window) and ensure that you have the following using statements:

```

using Microsoft.WindowsAzure.MobileServices;
using System.Windows.Threading;
using System.Device.Location;
using Microsoft.Phone.Maps.Controls;
using System.Windows.Shapes;
using System.Windows.Media;
using Windows.Networking.Proximity;

```

### Map

First, we'll need a function to center the map on the current user's location. To do this, we'll get the current location with our `locationWrapper`, then use `Map.Center` and `Map.ZoomLevel` to center the map.

```

public async void centerMap()
{
    string currLoc = await locationWrapper.GetCurrentLocation();
    Map.Center = new GeoCoordinate(Convert.ToDouble(currLoc.Split(':')[0]),
Convert.ToDouble(currLoc.Split(':')[1]));
    Map.ZoomLevel = 17;
}

```

When the page starts, our app should center the map, start the timer, and subscribe for the winning NFC tag. Replace the default `Tagger` method with:

```

public Tagger()
{
    InitializeComponent();
    centerMap();
    DispatcherTimer timr = new DispatcherTimer();
    timr.Interval = new TimeSpan(0, 0, 30);
    timr.Tick += timr_Tick;
    timr.Start();
    ProximityDevice.GetDefault().SubscribeForMessage("Windows.GPSTagWin",
MessageReceived);
}
private void MessageReceived(ProximityDevice sender, ProximityMessage message)

```

```
{
    sender.StopSubscribingForMessage(message.SubscriptionId);
    App.RootFrame.Navigate(new Uri("/Tagee.xaml"));
}
```

This code will center the map, set the timer to go off every thirty seconds, and subscribe for a the "Windows.GPSTagWin" message, then navigate to **Tagee.xaml** when it's encountered.

Finally, we'll add `timr_Tick`, which will add the other players to the map.

```
async void timr_Tick(object sender, EventArgs e)
{
    Map.Layers.Clear(); //let's clear out the map first
    IMobileServiceTable<Checkin> CheckinTable = App.MobileService.GetTable<Checkin>();
    var checkinsForThisGame = await CheckinTable.Where(a => a.gameId ==
App.gameId).ToListAsync();
    //http://www.developer.nokia.com/Community/Wiki/What%27s_new_in_Windows_Phone_8#Maps
    foreach (var checkin in checkinsForThisGame)
    {
        Map.Layers.Add(new MapLayer()
        {
            new MapOverlay()
            {
                GeoCoordinate = new
GeoCoordinate(Convert.ToDouble(checkin.location.Split(':')[0]),
Convert.ToDouble(checkin.location.Split(':')[1])), Content = new Ellipse {
                Fill = new SolidColorBrush(Colors.Red), Width = 40, Height = 40
            }
        }
    });
}
```

`timr_Tick` first clears all layers on the map control. Next, it uses Azure Mobile Services to get all the checkins for this game as a list, and cycles through them with a `foreach` loop. For each of those checkins, it adds a new layer onto the map with the locations held in the mobile service checkin as a 40 pixel red circle.

## JoinGame

Now that we have the ability to start a game, we need to add the ability to join one, too. The `JoinGame` page creates a new proximity device, subscribes to handle our NFC tag, and (when the tag is received) adds a new check in to the table indicating that they've joined the game.

### Using Statements

Open `JoinGame.xaml.cs` and ensure that you have the following using statements:

```
using Windows.Networking.Proximity;
using Microsoft.WindowsAzure.MobileServices;
```

### NFC

Replace the current `JoinGame()` function with:

```
public JoinGame()
{
```

```

InitializeComponent();
ProximityDevice device = ProximityDevice.GetDefault();
if (device != null)
{
    device.SubscribeForMessage("Windows.GPSTagGame", MessageReceived);
}
}

```

Next, add the MessageReceived handler which will be called when a phone is tapped to the current phone running our app.

```

private async void MessageReceived(ProximityDevice sender, ProximityMessage message)
{
    IMobileServiceTable<Checkin> CheckinTable = App.MobileService.GetTable<Checkin>();
    sender.StopSubscribingForMessage(message.SubscriptionId);
    string sent = message.DataAsString.Replace("\0", "");

    List<Checkin> playerIds = await CheckinTable.OrderByDescending(a =>
a.playerId).ToListAsync();
    App.playerId = playerIds[0].playerId + 1;

    App.gameId = Convert.ToInt32(sent);
    Checkin beginingCheckin = new Checkin();
    beginingCheckin.playerId = App.playerId;
    beginingCheckin.gameId = App.gameId;
    beginingCheckin.Comments = "Player Init";
    var loc = await locationWrapper.GetCurrentLocation();
    if (loc != "0:0")
    {
        beginingCheckin.location = loc;
        await CheckinTable.InsertAsync(beginingCheckin);
        App.RootFrame.Navigate(new Uri("/Tagger.xaml"));
    }
    else
    {
        MessageBox.Show("Something went wrong and we can't continue.");
    }
}
}

```

Notice that we're using the same tactic to get a unique player ID value as we did in StartGame above.

## Taggee

For the purpose of this tutorial, **Taggee.xaml** doesn't have a UI - it simply updates the checkin table every time the user moves 10 meters, and emits an NFC tag which allows the tagger to win. A production app would include the ability to end a game, message the tagger, and more.

### Using Statements

Open **Taggee.xaml.cs** and ensure that you have the following using statements:

```

using Windows.Devices.Geolocation;
using Microsoft.WindowsAzure.MobileServices;
using Windows.Networking.Proximity;

```

### Watching for Location Changes

First, add the `StartWatching` function which will automatically call a handler if the user moves 10 meters.

```
private void StartWatching()
{
    //http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj247548(v=vs.105).aspx
    Geolocator locator = new Geolocator();
    locator.DesiredAccuracy = PositionAccuracy.High;
    locator.MovementThreshold = 10;
    locator.PositionChanged += locator_PositionChanged;
}
```

Next, add the handler, which will add a new check in when the user moves 10 meters:

```
async void locator_PositionChanged(Geolocator sender, PositionChangedEventArgs args)
{
    IMobileServiceTable<Checkin> CheckinTable = App.MobileService.GetTable<Checkin>();
    List<Checkin> myCheckins = await CheckinTable.Where(a => a.playerId ==
App.playerId).ToListAsync();
    foreach (Checkin checkin in myCheckins)
    {
        await CheckinTable.DeleteAsync(checkin);
    }

    Checkin newCheckin = new Checkin();
    newCheckin.playerId = App.playerId;
    newCheckin.gameId = App.gameId;
    var loc = await locationWrapper.GetCurrentLocation();
    if (loc != "0:0")
    {
        newCheckin.location = loc;
        await CheckinTable.InsertAsync(newCheckin);
    }
    else
    {
        MessageBox.Show("Something went wrong and we can't continue.");
    }
}
```

Finally, call the `StartListening` method and start the NFC tag when the page is navigated to:

```
public Tagee()
{
    InitializeComponent();
    StartWatching();
    ProximityDevice.GetDefault().PublishBinaryMessage("Windows.GPSTagWin",
BufferFuncs.GetBufferFromString("WINNING"), MessageSent);
}
private void MessageSent(ProximityDevice sender, long messageId)
{
    sender.StopPublishingMessage(messageId);
    App.RootFrame.Navigate(new Uri("/Tagger.xaml"));
}
```

This code will set an NFC tag to emit "WINNING," and when the message is sent it will automatically navigate to the **Tagger**

page.

## Recap

In the case that something's still confusing to you about how and why we did things in the code, here's a recap of the technologies we used in our app.

### Windows Azure Mobile Services

Windows Azure Mobile Services essentially provides a wrapper for a cloud-hosted database. We signed up for the 3-month trial available on their website, and proceeded to create a table for our checkin type (note that when you create a table, the table name should match the name of your data type). The first thing you'll want to do (after using `Microsoft.WindowsAzure.MobileServices;`) when interacting with Mobile Services is to create an `IMobileServiceTable`, like so:

```
IMobileServiceTable<Checkin> CheckinTable = App.MobileService.GetTable<Checkin>();
```

Insert, update, and delete actions performed on an `IMobileServiceTable` are synced with the server in real-time. You can also use LINQ queries like `.where()` on an `IMobileServiceTable` to easily narrow down what you're looking for.

### Location Data

Using location data in your Windows Phone app helps make it's content more relative to the user, and can be especially important in apps (like ours) that use the user's location as an integral part of the application. You can use `windows.Devices.Geolocation` to get the current latitude and longitude, along with enabling a threshold which will automatically alert your app every time the user moves a specific amount of meters.

One important thing to note about location in Windows Phone is that you should make sure that you have your user's consent before using their location. The wrapper we developed above handles that for us, so feel free to use the `GetConfirmation` function in your own app.

Here's how to get the current user location on a Windows Phone:

```
Geolocator locator = new Geolocator();
locator.DesiredAccuracyInMeters = 1; //we want this as accurate as possible
Geoposition geoposition = await locator.GetGeopositionAsync(
    maximumAge: TimeSpan.FromMinutes(5),
    timeout: TimeSpan.FromSeconds(10)
);
var Latitude = geoposition.Coordinate.Latitude;
var Longitude = geoposition.Coordinate.Longitude;
```

If you would like to perform an action every time the user moves so many meters, you can use the following code:

```
Geolocator locator = new Geolocator();
locator.DesiredAccuracy = PositionAccuracy.High;
locator.MovementThreshold = 10; //change this to your desired movement threshold
locator.PositionChanged += locator_PositionChanged;
```

### <Maps /> Control

The Nokia `<Maps />` control is new to WP8, and replaces Bing Maps as the default mapping control for WP8 apps. You can also add layers onto the app to indicate, for instance, the position of other players in a game.

Here's how to center and zoom a `<Maps />` control:

```
string currLoc = await locationWrapper.GetCurrentLocation();
Map.Center = new GeoCoordinate(Convert.ToDouble(currLoc.Split(':')[0]),
    Convert.ToDouble(currLoc.Split(':')[1]));
```

```
Map.ZoomLevel = 17;
```

Note that we're using the `locationWrapper` we developed earlier, see above for how to use GPS in WP8.

If you would like to add a layer to the map, here's how you would do it:

```
double latitude = 0;
double longitude = 0;
Map.Layers.Add(new MapLayer()
{
    new MapOverlay()
    {
        GeoCoordinate = new GeoCoordinate(latitude, longitude, Content = new Ellipse {
            Fill = new SolidColorBrush(Colors.Red), Width = 40, Height = 40
        })
    }
});
```

You can also use `Map.Layers.Clear` to clear all current layers from the map.

## NFC

NFC (near-field communication) is another newly supported feature in WP8 which allows you to send and retrieve small amounts of data between two phones. You should use NFC instead of Bluetooth when you only need to transmit a small amount of data, you don't require extended two-way communication, and the app allows for the phones to touch each other. Typically, NFC can work within a range of no more than a few centimeters or lower. In our GPSTag app, we create an NFC tag that can be read by other phones. To do this (after using `Windows.Networking.Proximity` and using `Windows.Storage.Streams`, we used an elongated version of the following code:

```
private IBuffer GetBufferFromString(string str)
{
    using (var dw = new DataWriter())
    {
        dw.UnicodeEncoding = UnicodeEncoding.Utf16LE;
        dw.WriteString(str);
        return dw.DetachBuffer();
    }
}
long messageID = -1;
ProximityDevice device = ProximityDevice.GetDefault();
private async void StartNewGame()
{
    if (device != null)
    {
        messageID = device.PublishBinaryMessage("Windows.GPSTagGame",
        GetBufferFromString("Tag content goes here"), MessageSent);
    }
    else
    {
        MessageBox.Show("Sorry, your phone doesn't support NFC.");
    }
}
private void MessageSent(ProximityDevice sender, long messageid)
{
    MessageBox.Show("The message was succesfully read by another phone. Now ending the message...");
}
```

```
device.StopPublishingMessage(messageID);  
}
```

To read a tag, you need to subscribe to it and then create a handler for the tag. To do that, you would use:

```
public void StartSubscribing()  
{  
    ProximityDevice device = ProximityDevice.Default();  
    if (device != null)  
    {  
        device.SubscribeForMessage("Windows.GPSTagGame", MessageReceived);  
    }  
}  
private async void MessageReceived(ProximityDevice sender, ProximityMessage message)  
{  
    sender.StopSubscribingForMessage(message.SubscriptionId);  
    string messageAsString = message.DataAsString.Replace("\0", "");  
}
```

## Finished!

---

After completing this tutorial, you should have a basic understanding of concepts like how to use NFC, Windows Azure, and GPS data in your Windows Phone 8 app. For further reading, explore the sites listed below.

## See Also/Bibliography

---

- [How to continuously track the phone's location for Windows Phone 8](#) (MSDN)
- [What's new in Windows Phone 8](#) (Nokia Developer Wiki)
- [How to get the phone's current location for Windows Phone 8](#) (MSDN)
- [Windows Phone 8: Networking, Bluetooth, and NFC Proximity for Developers](#) (Microsoft BUILD Conference)