

# How to Store Application Data on NFC Tags

This article explains what is the best way to store custom application data on an NFC tag, also considering cross-platform compatibility and the ability to launch your app.

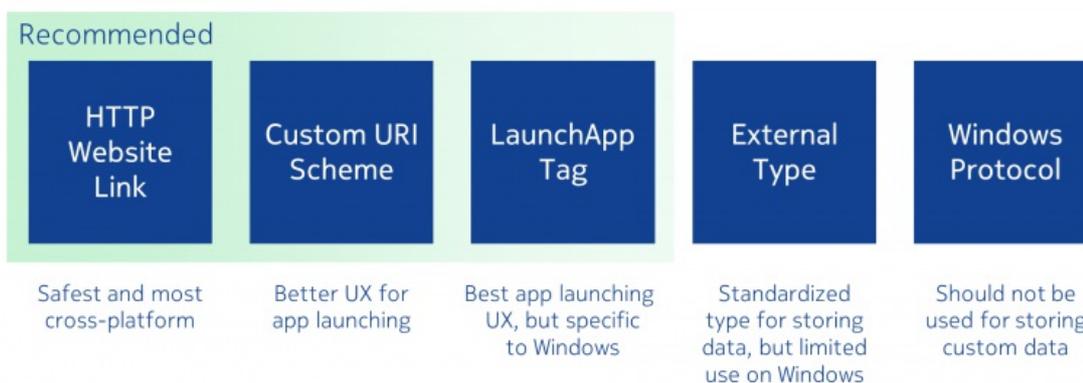
## Introduction



If you want to work with NFC tags, the most important decision is what kind of content to write on the tags. NFC is a complex technology. It's easy to choose a wrong or sub-optimal type; especially as many tutorials on the web suggest writing data in a way that limits your cross-platform compatibility.

This article provides an overview on the different content types to write, using Windows (Phone) 8 and its APIs as base for explanations and code examples. The following picture highlights the most relevant advantages and disadvantages:

## Storing App Data on NFC Tags



## NDEF as Base Layer

NFC tags can contain any kind of binary content - but to ensure that every device can read the tags, it is usually the best choice to write a standardized message based on the [NFC Data Exchange Format \(NDEF\)](#).

NDEF essentially adds a standardized header to your contents - the header contains information about what kind of content to expect as payload. The reading device then knows how to further handle the tag contents. Therefore, if you'd like other devices to read your tag contents, you need to follow the NDEF standard. In several use cases, this is not desirable, and the tag contents are not formatted according to NDEF standards - for example the case with many access badges or credit cards, which are not intended to be read by mobile devices out of the box.



**Note:** Windows 8 and Phone 8 only offer the possibility to work with NDEF data - the platform currently does not allow reading or writing data that is not part of an NDEF message.

## Content Types for Storing App Data

But even within the limits of NDEF, several different content types exist - each with their different uses and advantages. Some of them can be used to launch your app, some cannot be recognized by phones by default, and some have issues with cross-platform compatibility.

## Windows Protocol - usually should be your last choice



**Warning:** Only use the Windows Protocol (`Windows:WriteTag`) if you really know what you are doing and if you are using this record type on purpose.

Tapping a tag written using `Windows:WriteTag` will not trigger any kind of default action on a phone - the phone detects the message, but does not react to it. Therefore, you can't use this type to launch your application, which is commonly desirable when using NFC tags. It is possible to read these tags from within your application if it is already open.

Many Windows Proximity tutorials suggest writing data to tags using the `Windows:WriteTag` format - also the name of this type suggests that this type would write a standard tag. To write such a tag, code like the following is commonly used in tutorials and documentation:

```
var dataWriter = new DataWriter { UnicodeEncoding = Windows.Storage.Streams.UnicodeEncoding.Utf8 };
dataWriter.WriteString("Custom Data");
_device.PublishBinaryMessage("Windows:WriteTag.MyData", dataWriter.DetachBuffer(), MessageWrittenHandler);
```

The `dataWriter` contains the contents to be written to the tag; the type name after `Windows:WriteTag` (in this example `MyData`) defines the custom

subtype.

## Issues with the Windows Protocol

The `Windows:WriteTag` method will actually write an NDEF message with type name format `0x03` = Absolute URI to a tag. The NFC specifications require that the type name itself (in the above example `MyData`) is formatted according to absolute-URI construct defined by [RFC 3986](#).

URIs should identify a resource - via name, location or any other characteristic. It is recommended that the type follows URI scheme standards. The definition in [RFC 3986](#) describes URIs to start with a scheme name - e.g., "<http://www.ietf.org/rfc/rfc2396.txt>", "<mailto:John.Doe@example.com>" or "<tel:+1-816-555-1212>".

Obviously, our `MyData` type is not a valid URI scheme.

The [MSDN documentation](#) further suggests that the contents of the Windows type contain binary data.

The logical issue that arises is that any reading device gets an absolute URI that does not follow the standards and thus can't be handled, plus as the payload (= content) of the message some arbitrary binary data. As a consequence, you will usually run into problems if using the Windows Protocol for writing tags - phones can't handle them, and might even reject tags with a type that does not follow URI standards. Therefore, only use the Windows Protocol if you really know what you are doing and if you are using this record type on purpose.

## Writing Binary Data

According to the NFC Forum specifications, it is recommended to use an external type name (type name format = `0x04`) to store custom binary data (and not the Absolute URI type as used by the Windows Protocol, which should identify a resource as described above).

Through this external type, you can self-allocate a name space to be used for own purposes. The type name must be formed "by taking the domain name of the issuing organization, adding a colon, and then adding the type name as managed by the organization." (taken from the specification).

A valid type name would therefore be: `my.com:xx`.

The payload of the external record type can be anything, as defined by your application / organization.

## Issues with the External Type Name

While you can use external type names with Symbian and MeeGo, and even associate your application with your type name, this type is not well supported by Windows (Phone) 8.

Windows Phone will again detect the tag, but not react on it. Therefore, you can read messages formatted according to the external type name with your app, but you cannot automatically launch your app through this type. Additionally, the Proximity APIs only provide means to write completely custom binary data to tags, but do not help to wrap your data into an NDEF message formatted according to the external type name.

To do so, you would need to implement the NFC Forum standards manually, and shuffle a lot of bits and bytes around in order to create valid NDEF headers.

Luckily, you can use the open source [NDEF Library for Proximity APIs](#), which will wrap your binary data into a correct NDEF message. Using the library, you would write the message to a tag like this:

```
var externalRecord = new NdefRecord(NdefRecord.TypeNameFormatType.ExternalRtd, new[] { (byte)'m', (byte)'y', (byte)'.', (byte)'c', (
externalRecord.Payload = new[] {(byte) 'x'};
var ndefMsg = new NdefMessage {externalRecord};
_device.PublishBinaryMessage("NDEF:WriteTag", ndefMsg.ToArray().AsBuffer());
```

## Writing URIs that Contain Data

In general, the most compatible NFC tag content is a URL record - one of the well-known types that every NFC Forum compatible device should understand (for the technically interested: type name format: `0x01`, type name: `U`). With a URL, you can do two things:

**Link to your website:** e.g., <http://www.nfcinteractor.com/download/>

The page then contains further information about your app, download links, and a link to open the app. It's recommended to determine the platform on the website, to present the app download link for the right platform to the user. The link to launch the app would be formatted according to the custom URI scheme described below.

To embed custom app data, send it to the website as a parameter, which can then further process it and embed the data into the link to launch the app. E.g., <http://www.nfcinteractor.com/download/?mode=compose>

**Custom URI scheme:** e.g., `nfcinteractor:compose`

Windows Phone and Android allow your application to register for a custom URI scheme. When the user taps the tag, the phone will automatically launch the app registered for that scheme. In the example above, the scheme name would be `nfcinteractor:`, the app data `compose`.

In case the app is not yet installed, the user is sent directly to the store, where he can download your app.

A disadvantage of the custom URI scheme is that also your competitor could register for that URI scheme. If the user taps the tag and doesn't have the app installed yet, he would get the option to install both your and your competitor's app.

When to use a website, and when to use a custom URI scheme? Both approaches have unique advantages:

### Website link

- Better usage tracking: you can see how many people tapped your tag and followed the link, and gather statistics about the operating systems used.
- Presenting more information: on your website, you can present additional information about why the user should proceed to download your app.
- Safe option: a website URL works even on phones where you don't have a specific app available.

#### Custom URI scheme

- Better user experience: the tag directly launches the app, without the need to go to a website in-between (especially cumbersome if the app is already installed).

#### Which type should I choose?

There is no clear winner from those two options; it depends on your usage scenario. If you're placing your tags in a public place where you will get a lot of first-time users, it's probably best to use the normal website link. If you expect most of the users to have the app already installed, it's better to use the custom URI scheme.

To write a URL to an NFC tag, use the following code - it works for both the custom URI scheme as well as for HTTP links:

```
var dataWriter = new Windows.Storage.Streams.DataWriter { UnicodeEncoding = Windows.Storage.Streams.UnicodeEncoding.Utf16LE};
dataWriter.WriteString("nfcinteractor:compose");
var dataBuffer = dataWriter.DetachBuffer();
_device.PublishBinaryMessage("WindowsUri:WriteTag", dataBuffer);
```

---

## LaunchApp Tags

Both Windows Phone and Android defined own record types which can be directly linked to an application.

For Android, you would typically use a URL record as the first NDEF record in the message, and as a second record the *Android Application Record* (AAR). The first URL record ensures cross-platform compatibility. This second record uniquely contains your package name, and directly opens your app. The application can then parse the URL and any custom app data contained in the URL link or parameters. To write such a multi-record tag, you should also use the [NDEF Library](#), as Windows Phone doesn't offer support for composing multi-record tags out of the box.

For Windows (Phone), it's a little more tricky. Microsoft has defined the *LaunchApp* type, which can contain custom app data and your app ID. However, the LaunchApp record needs to be first on the tag - meaning that this will cause compatibility problems on most other platforms, which do not understand the Microsoft specific record and then don't handle the tag at all. The article [How to Create Cross-Platform LaunchApp NFC Tags](#) contains instructions on how to work around this limitation; but in many cases, you'll be better off to use a URL record as described above, and only use the LaunchApp if you're working in a constrained scenario where you know that almost all of your customers will use Windows-based devices.

---

## Summary

Choosing the right content type for storing app data is a tricky choice - especially as you usually also want the tag to be able to discover / launch your app.

In most cases, the *Windows Protocol* is not the right choice for storing app data. The *external type* was standardized for custom app data scenarios, but is not well supported on the Windows platform. The *LaunchApp* record of Windows works very well in a Windows-only scenario, but is problematic for cross-platform use.

This leaves the option of storing a *HTTP URL* or a *custom URI scheme*.

For most flexibility and cross-platform use, the HTTP URL link to your website is the safest choice. The custom URI scheme is a compromise that works well on Windows and Android and directly launches your app, but already reduces the cross-platform compatibility.

Make your choice depending on how much you know about your target audience. For public tags that you place in the wild, it's usually best to link to your website, which then further handles platforms and can launch the app. For scenarios where you know that the majority of your users will use Windows or Android, and if you want to provide the best user experience to launch the app, use a custom URI scheme.

--ajakl 12:37, 16 December 2012 (EET)

