CHAPTER 1

Introduction

This book started life innocently enough in an email from Brian to Tom in March 2011. Brian thought it would be a good idea to add a couple of lines to the second edition of another book about near field communication, *Making Things Talk*, which we were working on at the time. There was already a chapter on radio frequency identification (RFID) in the book, so how hard could it be? Two and a half years later, we've learned a lot about NFC along the way and picked up an excellent and knowledgeable collaborator, Don Coleman, author of the NFC plug-in for PhoneGap.

Even though NFC has a lot of potential, most of the material written about it so far hasn't been written for the casual programmer. Everything out there assumed that if you wanted to know about NFC, you were prepared to do it from the silicon up. You had to understand the details of the various RFID specs involved, and you had to be prepared to write code that interpreted the byte stream from an NFC reader one byte at a time. While it's useful to understand that, we figured NFC would see wider use if programmers could concentrate on what they were using it for, rather than the low-level details. Don's PhoneGap library was the best tool we found to do just that. It lets you design NFC exchanges in the way we imagine the NFC forum designers intended: you think about the messages being exchanged and don't worry about the rest.

Most of this book is written in that spirit. You'll learn about the basics of the NFC Data Exchange Format (NDEF) by reading and writing messages from device to tag and from device to device. You'll see a few sample applications—some written for PhoneGap, some for Arduino, and some for Node.js—running on embedded devices like the Raspberry Pi and the BeagleBone Black. You'll learn some of the use patterns of NDEF, and you'll get a taste of how you might think about the physical interaction of NFC-driven applications.

The state of the art varies from platform to platform, however. Not everything that the NFC Forum specifications describe is accessible to the casual programmer on every platform yet. We've attempted to give you a roadmap in this book, particularly in the later chapters, as to what the current state of development is, and where there is still room for usability improvement.

We hope that this book will help the casual programmer get a sense of what can be done using NFC, and that it will inspire more professional developers to create simple-to-use tools to help spread its use.

Who This Book Is For

You don't have to be a trained professional programmer to read this book. We tried to write it for programming enthusiasts—people who've picked up some knowledge along the way, but maybe not in a formal learning setting. You won't learn to write enterprise-level code here, but you will get a practical introduction to what near field communication is and how to program applications using it on Android, Arduino, and embedded Linux.

We assume you have some familiarity with programming, however. You will want to be familiar with JavaScript and HTML for most of the examples in the book. You'll get introduced to a little C in the Arduino projects, but if you're familiar with JavaScript or Java, it will look familiar enough. For those latter projects, you should be a little familiar with electronics, but you don't have to be.

Recommended Reading

"What? I have to read other books in order to read this book?" No, but there are a few books that we found helpful in writing this one. We thought you might find them useful as well.

If you're new to JavaScript, read Douglas Crockford's *JavaScript: The Good Parts*. Come to think of it, read it if you're an old hand at JavaScript. It'll make you a better programmer. He explains the theoretical underpinnings of the language and the best use patterns clearly and definitively.

For PhoneGap and Android, the online Getting Started Guides are the most up-to-date references; see the PhoneGap developer portal and the Android developer site. For more in-depth introductions to Android, see *Professional Android 4 Application Development* or *Android Programming: The Big Nerd Ranch Guide*.

For an in-depth introduction to NFC from an engineering perspective, the NFC Forum Specifications are the original source material (we have reproduced some of them in Appendix A for handy reference). We also found *Professional NFC Application Development* by Vedat Coskun, Kerem Ok, and Busra Ozdenizci to be a good reference, particularly for experienced Java programmers. We have deliberately taken a more populist approach than that book, since many of our readers are hobbyists, hackers, and other self-identified dilettante programmers.

If you're new to Arduino, Massimo Banzi's *Getting Started with Arduino* is an excellent starting place. Tom Igoe's *Making Things Talk, 2nd Edition* is a good book for experienced programmers to learn about connecting Arduino projects to networks. Michael Margolis' *Arduino Cookbook* has some handy recipes for Arduino programs as well.

For an introduction to Node.js, which pops up later in this book, Brett McLaughlin's *What is Node?* is a nice essay-length introduction with no code. Manuel Kiessling's *The Node Beginner Book* and Pedro Teixeira's *Hands-On Node.js* are helpful and short guides to getting started with the actual code.

For a good introduction to the Raspberry Pi or the BeagleBone Black, which you'll encounter in Chapter 9, you can find material for getting started on Adafruit's tutorials. The books *Getting Started with Raspberry Pi* by Matt Richardson and Shawn Wallace and *Getting Started with BeagleBone* by Matt Richardson are also good introductions.

What's Covered in This Book

Chapter 2 gives you an introduction to near field communication (NFC) by comparing it to radio frequency identification (RFID). Simply put, NFC is a superset of RFID. It can do most things short-range RFID can do, and more. You'll get a preview of the most important terms, a look at the architecture of an NFC system, and learn what tools you need and where to get them.

Chapter 3 introduces you to PhoneGap and the NFC plug-in for PhoneGap. You'll install the tools necessary to develop PhoneGap applications for Android and build and run your first couple of applications. By the end of this chapter, you'll have read your first NFC tag using an Android device.

Chapter 4 is an in-depth overview of the NFC Data Exchange Format (NDEF). You'll learn how it's structured and see it in practice by writing an application that performs the same basic task using different types of NDEF records, to see how each record type affects user interaction on Android.

Chapter 5 covers how to listen for NDEF messages on Android. You'll learn how to filter for different types of tags and messages, and how the Android Tag Dispatch system can be used to your advantage when developing NFC apps.

In Chapter 6, you'll build a full NFC application on Android that features a full user interface, audio playback, and control of web-connected lighting, all mediated by NFC tags. The goal of this chapter is to show you how to plan the interaction design and data formatting of an application to best take advantage of NFC.

Chapter 7 brings another platform into play: the Arduino microcontroller development platform. You'll learn how to read and write NDEF messages using the Arduino NDEF library. You'll also develop another full application using Arduino and Node.js.

Chapter 8 introduces you to peer-to-peer exchanges using NFC on Android. You'll learn how the record types you're exchanging through peer-to-peer affect the receiving device, and you'll learn about how NFC can negotiate the handoff of larger exchanges to alternate carriers like Bluetooth and WiFi.

Chapter 9 gives you the state of the art on NFC development on embedded Linux platforms using the Raspberry Pi and BeagleBone as examples. You'll get an understanding of what's possible on embedded Linux, and see a few sample applications in Node.js. There's still a lot of room for usability improvements in this context, so be warned that this chapter is not for the technically timid. You'll want some familiarity with the Linux command-line interface to get the most out of this chapter. This is where some of the most exciting possibilities for NFC use lie, though, so it's good territory to know.

What You'll Need

To do the exercises in this book, you'll need some hardware and software. All of the software is free, fortunately. The most expensive piece of hardware used here is an NFC-enabled Android device. The following sections list what you'll be using.

Hardware

To follow along with the book overall, you'll need the following hardware:

- An Android NFC-enabled device
- Several NFC-compatible tags (check compatibility with your devices; "Device-to-Tag Type Matching" on page 19 includes a chart showing which devices work with which tag types)

For Chapter 6, you'll need:

- A Philips Hue lighting system
- A Bluetooth Music Receiver (e.g., Belkin Bluetooth Music Receiver or HomeSpot's NFC-enabled Bluetooth Audio Receiver)

For Chapter 7, you'll need:

- An Arduino Uno microcontroller, available from many outlets, including Arduino, Adafruit, Seeed Studio, RadioShack and others
- An NFC Shield (you can use Adafruit's PN532 NFC/RFID Controller Shield for Arduino or Seeed Studio's NFC Shield or NFC Shield v2.0)
- If you're using the Adafruit shield, you may want to get some shield stacking headers from Adafruit as well.

- A solenoid-driven door lock, 12V or less. We used an Amico 0837L DC 12V 8W open frame type solenoid for electric door lock bought on Amazon, but you can also get solenoids from other retailers. Adafruit sells a similar lock-style solenoid and Seeed Studio sells several models, so if you're ordering a shield from them, you can get a solenoid from them as well.
- A TIP120 Darlington transistor
- A 12V, 1000mA power supply, with 2.1mm ID, 5.5mm OD, center-positive connector to power the solenoid circuit
- Jumper wires or 22AWG solid-core wire
- Two LEDs (one red, one green). These are available from any electronics retailer, but for reference, check out Adafruit's red LED pack or green LED pack.
- Two 220Ω resistors for the LEDs

For Chapter 9, you'll need:

- A BeagleBone Black or Raspberry Pi embedded Linux microcontroller
- 1 amp or greater power supply for your board
- An SLC3711 Contactless USB Smart Card Reader
- Optional but useful:
  - A USB WiFi adapter for your board (Adafruit's Miniature WiFi (802.11b/g/n) Module works well)
  - A USB A-to-A extender for your NFC adapter
  - A USB to TTL serial cable—debug/console cable

Table 1-1 lists the electronics components for this book with part numbers from some of the electronics distributors we use regularly, in case you want alternatives to those listed previously.

Table 1-1. Electronic components used in this book

Part	MakerShed	Jameco	Digikey	SparkFun	Adafruit	Farnell	Arduino	Seeed
220Ω resistor		690700	220QBK-ND			9337792		
Solderless breadboard	MKEL3	20723	438-1045-ND	PRT-00137	64	4692810		STR101C2M or STR102C2M
Red hookup wire	MKSEEED3	36856	C2117R-100-ND	PRT-08023		1662031		
Black hookup wire	MKSEEED3	36792	C2117B-100-ND	PRT-08022		1662027		
Blue hookup wire	MKSEEED3	36767				1662034		

Part	MakerShed	Jameco	Digikey	SparkFun	Adafruit	Farnell	Arduino	Seeed
12V 1000mA DC power supply (or equivalent)		170245		TOL-00298	798	636363		
Arduino Uno rev3 Microcontroller Module	MKSP11	2121105	1050-1017-ND	DEV-09950	50	1848687	A000046	ARD132D2P
Green LED	MKEE7	333227		COM-09592		1334976		
Red LED	MKEE7	333973		COM-09590		2062463		
Blue LED		2006764		COM-00529		1020554		
Yellow LED	MKEE7	34825		COM-09594		1939531		
MIFARE RFID tag	MKPX4			SEN-10128				
NFC shield	MKAD45				789			SLD01097P
TIP120 Darlington transistor		10001_ 10001_ 32993_-1	TIP120-ND		976	9294210		
3.3V USB/TTL serial debug cable				DEV-09717	954			
BeagleBone Black	MKCCE3	2176149	BB-BBLK-000-ND		1278	2291620		ARM00100P
Raspberry Pi Model B	MKRPI2			DEV-11546	998	43W5302		

Software

To follow along with the book overall, you'll need the following software:

- The Android software development kit (see "Setting Up the Development Environment" on page 24)
- The Cordova CLI, your toolbox for PhoneGap (see "Install Cordova CLI for PhoneGap" on page 28)
- Node.js and the Node Package Manager (npm)
- A text editor (we like Sublime Text 2 as a cross-platform GUI editor, but you can use anything that can generate a plain-text file)

For Chapter 6, you'll need:

- The Zepto jQuery library, available from Zepto.js

For Chapter 7, you'll need:

- The Arduino IDE

- The Seeed-Studio PN532 library
- The Arduino NDEF library, available from Don Coleman's GitHub repository
- The Time library for Arduino by Michael Margolis

Don't worry about setting all this up right now; we'll let you know when you need to install a piece of software.

Other Useful NFC Apps

The following will be useful throughout the course of the book:

- NFC TagInfo by NXP allows you to read any NFC or Mifare tag and examine the NDEF record on it.
- NFC TagWriter by NXP allows you to do many of the same things as TagInfo. It can also write tags, and unformat tags, which is really handy.
- NFC Research Lab's NFC TagInfo will show you all the info about a given tag. It's more advanced than NXP's TagInfo, in that it will also allow you to see a memory dump from the tag. It's invaluable for troubleshooting your applications.

For Chapter 4, you'll need:

- Trigger by TagStand
- NFC Writer, also by TagStand
- TecTiles by Samsung (functional in the United States and Canada only)
- App Lancher NFC Tag Writer [sic] by vvakame
- A Foursquare account if you don't have one already, and Foursquare for Android

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/tigoe/beginningnfc*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Beginning NFC: Near Field Communication with Arduino, Android, and PhoneGap*, by Tom Igoe, Don Coleman, and Brian Jepson. Copyright Tom Igoe, Don Coleman, and Brian Jepson 2014 978-1-4493-6307-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/beginning-nfc*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

We've received generous assistance from many people and organizations during the writing of this book. The PhoneGap-NFC plug-in was Kevin Griffin's brainchild; he and Don wrote the first version of it and presented it at PhoneGap Day 2011. Kevin Townsend of Adafruit has been an invaluable resource for his in-depth knowledge of NXP's software and hardware. Yihui Xiong of Seeed Studio and author of the Seeed Arduino NDEF library, was crucial to the success of Chapter 7. Philippe Teuwen made fast patches to libfreefare, clearing roadblocks in Chapter 9. Derek Molloy's pages on the BeagleBone

Black were a handy resource. The images in this book are better thanks to Jody Culkin and Fritzing.org. Early readers Ben Light, Sae Huh, Gabrielle Levine, Alex Kauffmann, Fil Maj, and Dominick Gruntz offered valuable feedback.

This book was made possible by the patient support of our employers and colleagues at NYU's Interactive Telecommunications Program (ITP), Chariot Solutions LLC, Maker Media, and Arduino. We would also like to acknowledge the even more patient support of our families and partners.

CHAPTER 2

NFC and RFID

Radio frequency identification (RFID) is becoming commonplace in everyday life these days. From tap-and-go payment cards and transit passes to E-ZPass devices used on toll roads to the tags stuck on and sewn into consumer goods to manage inventory and deter theft, most of us encounter RFID tags at least a few times a week and never think about what can be done with this technology.

In the past few years, a new term has started to bubble up in connection with RFID: *near field communication* (NFC). Ask your average techie what it is and you'll probably hear "Oh, it's like RFID, only different." Great, but how is it different? RFID and NFC are often conflated, but they're not the same thing. Though NFC readers can read from and write to some RFID tags, NFC has more capabilities than RFID, and enables a greater range of uses. You can think of NFC as an extension of RFID, building on a few of the many RFID standards to create a wider data exchange platform.

This book aims to introduce you to NFC and its capabilities in a hands-on way. Following the exercises in these chapters, you'll build a few NFC applications for an NFC-enabled Android device and for an Arduino microcontroller. You'll learn where RFID and NFC overlap, and what you can do with NFC.

What's RFID?

Imagine you're sitting on your porch at night. You turn on the porch light, and you can see your neighbor as he passes close to your house because the light reflects off him back to your eyes. That's passive RFID. The radio signal from a passive RFID reader reaches a tag, the tag absorbs the energy and "reflects" back its identity.

Now imagine you turn on your porch light, and your neighbor in his home sees it and flicks on his porch light so that you can see him waving hello from his porch. That's active RFID. It can carry a longer range, because the receiver has its own power source, and can therefore generate its own radio signal rather than relying on the energy it absorbs from the sender.

RFID is a lot like those two porches. You and your neighbor know each other's faces, but you don't really learn a lot about each other that way. You don't exchange any meaningful messages. RFID is not a communications technology; rather, it's designed for identification. RFID tags can hold a small amount of data, and you can read and write to them from RFID readers, but the amount of data we're talking about is trivial, a thousand bytes or less.

What's NFC?

Now imagine another neighbor passes close, and when you see her, you invite her on to the porch for a chat. She accepts your invitation, and you sit together, exchange pleasantries about your lives, and develop more of a relationship. You talk with each other and you listen to each other for a few minutes. That's NFC.

NFC is designed to build on RFID by enabling more complex exchanges between participants. You can still read passive RFID tags with an NFC reader, and you can write to their limited amount of memory. NFC also allows you to write data to certain types of RFID tags using a standard format, independent of tag type. You can also communicate with other NFC devices in a two-way, or duplex, exchange. NFC devices can exchange information about each other's capabilities, swap records, and initiate longer term communications through other means.

For example, you might tap your NFC-enabled phone to an NFC-enabled stereo so that they can identify each other, learn that they both have WiFi capability, and exchange credentials for communication over WiFi. After that, the phone will start to stream audio over WiFi to the stereo. Why doesn't the phone stream its audio over the NFC connection? Two reasons: first, the NFC connection is intentionally short range, generally 10cm or less. That allows it to be low-power, and to avoid interference with other radios built into devices using it. Second, it's relatively low-speed compared to WiFi, Bluetooth, and other communications protocols. NFC is not designed to manage extended high-speed communications. It's for short messages, exchanging credentials, and initiating relationships. Think back to the front porch for a moment. NFC is the exchange you have to open the conversation. If you want to talk at length, you invite your neighbor inside for tea. That's WiFi, Bluetooth, and other extended communications protocols.

What's exciting about NFC is that it allows for some sophisticated introductions and short instructions without the hassle of exchanging passwords, pairing, and all the other more complicated steps that come with those other protocols. That means that when you and your friend want to exchange address information from your phone to his, you can just tap your phones together. When you want to pay with your Google Wallet, you can just tap as you would an RFID-enabled credit card.

When you're using NFC, your device doesn't give the other device to which it's speaking access to its whole memory—it just gives it the basics needed for exchange. You control what it can send and what it can't, and to whom.

How RFID Operates

An RFID exchange involves two actors: a *target* and an *initiator*. The initiator, a tag reader or reader/writer device, starts the exchange by generating a radio field and listening for responses from any target in the field. The target, a tag, responds when it picks up a transmission from an initiator. It will respond with a *unique identifier number* (UID). RFID has two *communication modes*: active and passive. *Passive RFID* exchanges involve a reader/writer and a tag that has no power source on board. The tags get their power from the energy of the radio field itself. It's generally a very small amount, just enough to send a signal back to the reader. *Active RFID* exchanges involve a target that's an independently powered device. Because the target is powered, its reply to the reader can travel a much greater distance. E-ZPass and other traffic ID systems use active RFID.

RFID tags have a small amount of memory on board, usually less than 1 kilobyte. An initiator device can read this data, and if it's a reader/writer device, it can write to the tag as well. This allows you to store small amounts of information associated with the card. For example, it's sometimes used in transit systems that use RFID, to keep track of how much value is left on the card. However, since RFID systems generally are networked to a database, it's more common to store a data record indexed by the tag's UID in a remote database, and store all information about the tag in that remote database.

RFID Standards

Contrary to popular belief, there is no single universally interoperable RFID protocol or technology. There are dozens. RFID standards are developed by the International Standards Organization (ISO), in conjunction with major participants in the RFID market. ISO works as a mediating body to help competitors in many different industries develop interoperable standards so that even when they compete, their technologies can sometimes work together. The various RFID standards define the radio frequencies used, the data transfer rates, the data formats, and more. Some of these standards define layers of a single interoperable stack, as you'll see with NFC. Other standards define a whole different class of applications. For example, the ISO-11784 standard was originally developed for animal tracking. It operates in frequencies between 129 and 139.4kHz, and its data format features fields suited to describing the animals to be tracked. You can also find EM4100 procotol readers and tags that operate in the 125kHz range. These are often used as proximity cards, and feature very limited information in their data protocol, usually just a UID. The ISO-14443 standards were developed for use with payment systems and smart cards. They operate at 13.56MHz. They include features in their data format for incrementing and decrementing values and for encrypting data, for example. Within the 14443 family, there are several different formats including Philips and NXP Mifare tags, Sony FeliCa tags, and NXP DESFire. ISO-14443A tags are compatible with NFC, so you'll see a lot of them in the pages that follow.

How NFC Operates

NFC can be thought of as an extension of RFID. NFC exchanges also involve an initiator and a target like RFID. However, it can do more than just exchange UIDs and read or write data to the target. The most interesting difference between RFID and NFC is that NFC targets are often programmable devices, like mobile phones. This means that rather than just delivering static data from memory, an NFC target could actually generate unique content for each exchange and deliver it back to the initiator. For example, if you're using NFC to exchange address data between two phones, the NFC target device could be programmed to only provide limited information if it's never seen this particular initiator before.

NFC devices have two *communications modes*. If the initiator always supplies the RF energy and the target gets powered by the initiator's field, they're said to be engaging in *passive* communication mode. If both target and initiator have their own energy sources, they're in *active* communication mode. These modes are the same as regular RFID communication modes.

NFC devices have three *operating modes*. They can be *reader/writers* that read data from a target and write to it. They can be *card emulators*, acting like RFID tags when they're in the field of another NFC or RFID device. Or they can operate in *peer-to-peer mode*, in which they exchange data in both directions.

NFC Data Exchange Format (NDEF)

Data exchanged between NFC devices and tags is formatted using the *NFC Data Exchange Format* (NDEF). This is a term you'll hear a lot; NDEF is one of the key advancements that NFC adds to RFID. It's a common data format that operates across all NFC devices, regardless of the underlying tag or device technology. Every NDEF message contains one or more NDEF records. Each record has a particular record type, a unique ID, a length, and a payload of data. There are a few well-known types of NDEF records. NFC-enabled devices are expected to know what to do with each of these types:

Simple text records
: These contain whatever text string you want to send. Text messages generally don't contain instructions for the target device. They also include metadata indicating the language and encoding scheme (e.g., UTF-8).

URIs
: These contain network addresses. An NDEF target device that receives a URI record is expected to pass that record to an application that can display it, such as a web browser.

Smart Posters
: These contain data you might attach to a poster to give it more information. This can include URIs, but might also contain other data, like a text message to be sent about the poster, telling your friends about it. A target device that receives a Smart Poster record might open a browser, SMS, or email application, depending on the message's content.

Signatures
: These provide a way to give trustworthy information about the origins of data contained in an NDEF record.

You can mix and match records in an NDEF message, or you can send only one record per message, as you choose.

NDEF is one of the important technical differences between RFID and NFC. Both NFC and many of the RFID protocols operate on 13.56MHz, but RFID tags do not have to format their data in NDEF format. The various RFID protocols do not share a common data format.

Think of NDEF messages like paragraphs and records like sentences: a paragraph is a discrete chunk of information that contains one or more sentences. A sentence is a smaller chunk of information that contains just one idea. For example, you might write a paragraph that indicates that you're having a birthday party, and gives the address. The NDEF message equivalent might contain a simple text record to describe the event, a Smart Poster record containing the physical address, and a URI for more information on the Web.

The Architecture of NFC

In order to understand NFC in depth, it helps to have a mental model of the architecture. There are several layers to consider. The lowest layer is the physical, namely your CPU and the radios that are doing the communication. In the middle, there are data packetization and transport layers, then data format layers, and finally, your application code. Figure 2-1 shows the various layers of the NFC stack.

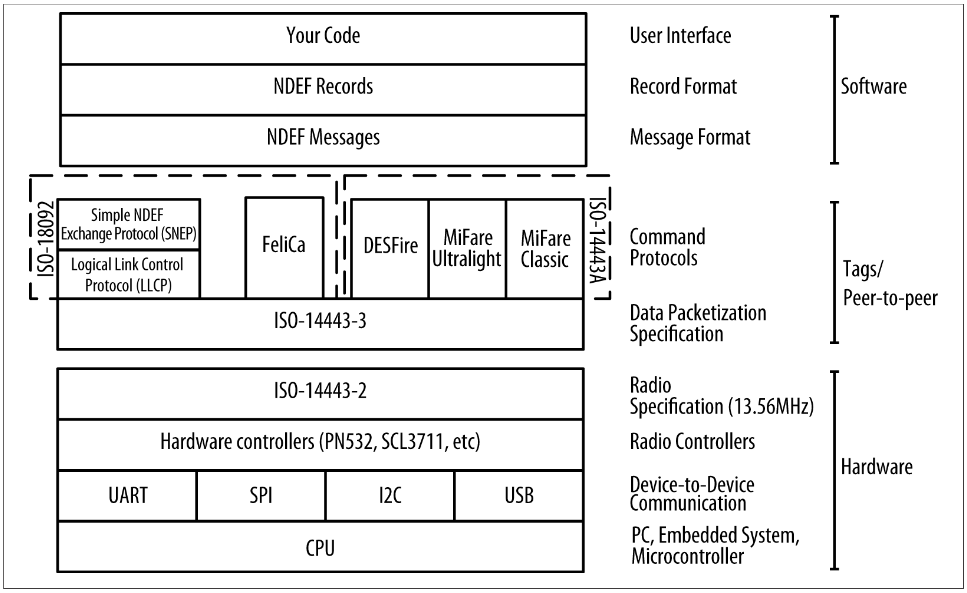


Figure 2-1. The NFC protocol stack

At the physical layer, NFC works on an RFID radio specification, ISO-14443-2, that describes low-power radios operating at 13.56MHz. Next comes a layer that describes the framing of data bytes sent over the radio, ISO-14443-3. Any of the radios you might use are separate hardware components, either inside your phone or tablet, or attachments to your microcontroller or personal computer. They communicate with the main processor of your device using one or more standard inter-device serial protocols: *universal asynchronous receive-transmit* (UART), *serial peripheral interface* (SPI), *inter-integrated circuit communication* (I2C), or *universal serial bus* (USB). If you're a hardware enthusiast, you probably know the first three, but if your focus is software, you may only know USB.

Above that are several RFID command protocols, based on two specifications. The original RFID control specification that NFC tag reading and writing is built on is ISO-14443A. The Philips/NXP Semiconductors Mifare Classic and Mifare Ultralight and NXP DESFire protocols are compatible with ISO-14443A. The NFC peer-to-peer exchange is built on the ISO-18092 control protocol. Sony FeliCa RFID cards and tags, mostly available in Japan, are based on this standard as well. You can read from and write to tags using these standards and still not be using NFC, however.

The first major difference between NFC and RFID is the peer-to-peer communications mode, which is implemented using the ISO-18092 standard. There are two protocols, a *logical link control protocol* (LLCP) and a *simple NDEF exchange protocol* (SNEP) that manage peer-to-peer exchanges. These are shown in the stack alongside the control protocols, since they operate on the same standard as one of them.

The second major difference between RFID and NFC, the NFC Data Exchange Format (NDEF), sits atop these control protocols. NDEF defines a data exchange in messages, which are composed of NDEF records. There are several different record types, which you'll learn more about in Chapter 4. NDEF makes it possible for your application code to deal with data from NFC tag reading and writing, peer-to-peer exchanges, and card emulation all the same way.

Most of this book focuses on the use of NDEF messaging. Although we'll discuss the lower layers so that you understand them, we think that the advantage to NDEF is that it frees you from thinking about them.

NFC Tag Types

There are four types of tags defined by the NFC forum, all based on the RFID control protocols described previously. There's a fifth that's compatible, but not strictly part of the NFC specification. Types 1, 2, and 4 are all based on ISO-14443A, and type 3 is based on ISO-18092. You can get tags from many vendors, and you've probably run across them in everyday life without knowing it. Their details are as follows:

Type 1

- Based on ISO-14443A specification.
- Can be read-only, or read/write capable.
- 96 bytes to 2 kilobytes of memory.
- Communication speed 106Kb.
- No data collision protection.
- Examples: Innovision Topaz, Broadcom BCM20203.

Type 2

- Similar to type 1 tags, type 2 tags are based on NXP/Philips Mifare Ultralight tag (ISO-14443A) specification.
- Can be read-only, or read/write capable.
- 96 bytes to 2 kilobytes of memory.
- Communication speed 106Kb.
- Anti-collision support.
- Example: NXP Mifare Ultralight.

Type 3

- These are based on the Sony FeliCa tags (ISO-18092 and JIS-X-6319-4), without the encryption and authentication support that FeliCa affords.

- Configured by factory to be read-only, or read/write capable.
- Variable memory, up to 1MB per exchange.
- Two communication speeds, 212 or 424Kbps.
- Anti-collision support.
- Example: Sony FeliCa.

Type 4

- Similar to type 1 tags, type 4 tags are based on NXP DESFire tag (ISO-14443A) specification.
- Configured by factory to be read-only, or read/write capable.
- 2, 4, or 8KB of memory.
- Variable memory, up to 32KB per exchange.
- Three communication speeds: 106, 212, or 424Kbps.
- Anti-collision support.
- Example: NXP DESFire, SmartMX-JCOP.

A fifth type, proprietary to NXP Semiconductors, is probably the most common in NFC use today:

Mifare Classic tag (ISO–14443A)

- Memory options: 192, 768, or 3,584 bytes.
- Communication speed 106Kbps
- Anti-collision support.
- Examples: NXP Mifare Classic 1K, Mifare Classic 4K, Mifare Classic Mini.

Where to Get Tags

Mifare and NFC tags are increasingly easy to get. To make the most of this book, you should have some NFC Forum tags and some Mifare Classic tags as well.

Many mobile phone stores now carry Samsung's TecTiles, which work well for many of the applications in this book. They're expensive, but convenient. The original TecTile tags are Mifare Classic, which work well on Arduino in Chapter 7 and the USB NFC reader on the BeagleBone Black and Raspberry Pi in Chapter 9. It's not clear, but it would appear that Samsung is abandoning TecTile in favor of TecTile 2. There is a general shift among hardware vendors away from Mifare Classic and toward the official NFC Forum tag types.

If you want tags in different form factors, Adafruit has a wide variety in their NFC category. SparkFun and Seeed Studio also have plenty of Mifare tags; search for "NFC" or "Mifare" and you'll find plenty.

You can also get tags from TagAge, but shipping can be slow from Finland, depending on your location. Customer service has been excellent for us, however.

Identive NFC can deliver tags overnight. They are a bit more expensive, but they arrive quickly and they have a cool android on them.

Device-to-Tag Type Matching

Not all NFC-compatible tags work with all NFC devices. In the course of working on this book, we've tested many different devices and tags, and have found some combinations work, and some don't. Though Table 2-1 isn't an exhaustive table, it's intended to give you enough information to choose your devices and tags appropriately.

The most common issue you'll run into is that some devices will read Mifare Classic tags, while others won't. Mifare Classic is not actually an NFC Forum standard tag type, and recently, major hardware vendors like Broadcom and Samsung have started to drop support for it on their devices. You're likely to get unpredictable results when you try to read Mifare Classic tags with some of these newer tags. For example, the Nexus 4 will read Mifare Classic as a generic tag, but the Samsung S4 (Google Edition) displays an error saying it can't handle the tag. When possible, we recommend going with one of the official NFC Forum tag types. In the chapters ahead, we'll make more specific recommendations where appropriate.

If you're coming to NFC because you're interested in using it for embedded hardware projects on the Arduino, Raspberry Pi, or BeagleBone Black (which you'll learn more about in Chapter 7 and Chapter 9), you should make sure you have both NFC tags and Mifare Classic tags. The libraries for those platforms were developed starting with Mifare Classic support, and as of this writing, they don't handle all the NFC Forum types. We've added Mifare Ultralight (NFC type 2) read support for the Arduino NDEF library (which Don wrote), and there is partial Ultralight and DESFire support (NFC types 2 and 4) in libfreefare, one of the libraries we'll show you in Chapter 9.

While we hope to see additions to those libraries to support all of the NFC Forum tag types, it's not there yet. Growing pains like these are common with emerging technologies, but so far, signs point to the industry moving toward a standardized set of tag types.

Table 2-1. Device-to-tag compatibility

Device	Type 1	Type 2 (Mifare Ultralight)	Type 3 (Sony FeliCa)	Type 4 (Mifare DESFire)	Mifare Classic
Samsung Galaxy S	Yes	Yes	Yes	Yes	Yes
Google Nexus S	Yes	Yes	Yes	Yes	Yes
Google Galaxy Nexus	Yes	Yes	Yes	Yes	Yes
Google Nexus 7 version 1	Yes	Yes	Yes	Yes	Yes
Google Nexus 7 version 2	Yes	Yes	Yes	Yes	No
Google Nexus 4 (phone)	Yes	Yes	Yes	Yes	No
Google Nexus 10 (tablet)	Yes	Yes	Yes	Yes	No
Samsung Galaxy S4 (tablet)	Yes	Yes	Yes	Yes	No
Samsung Galaxy SIII	Yes	Yes	Yes	Yes	Yes
Adafruit NFC Shield	No	Partial	No	No	Yes
Seeed Studio NFC Shield for Arduino	No	Partial	No	No	Yes
NFC USB Dongle (libnfc)	No	Partial	No	Partial	Yes

What You Can Do with NFC

There are many possible applications for NFC. Its tag emulation functionality is showing up in monetary transactions like Google Wallet, and in payment and ticketing systems for public transport. There are several mobile phone apps that allow you to save configuration data for your phone to a tag, and use the tag to automatically change the context of your phone (e.g., set the quiet settings for meetings, turn on WiFi and configure for a given network by tapping a tag). Home audio equipment is coming on the market that allows you to automatically pair your phone or tablet with your audio player or TV, to use the former as a remote control for the latter. Healthcare systems have implemented patient ID and record linking using NFC, and the NFC Forum recently released a Personal Health Device Communication (PHDC) technical specification to aid in that development. Inventory management can be improved by using NDEF records written to the goods being shipped, about their journey, shipping times, and so forth. Fleet tracking can also be enhanced using NFC tags and readers by storing route information, travel times, mileage, maintenance, and fuel information on tags, or transferring it from vehicle to mobile device through peer-to-peer.

In the course of working on this book, we've seen NFC-compatible tags in many locations, from New York City's Citi Bike key fobs to metro passes in China and Norway to hotel room keys, and more. Although not all of these applications are using the tags to the full potential NFC affords, they all could be enhanced or simplified through some of its features.

The combination of peer-to-peer messaging between devices with minimal negotiation, and a common data format between device and tag messages makes NFC a potentially powerful tool for data-intensive applications in the physical world.

Conclusion

Near field communication adds some promising new functionality to RFID technologies. Most notable of these is the NFC Data Exchange Format, NDEF, which provides a common data format across the four different NFC tag technologies. NDEF can be used for both tag-to-device data exchange and for device-to-device data exchange. This distinguishes NFC as more than just an identification technology; it is also a useful technology for short data exchanges. The method by which NFC devices and tags interact—by means of a single tap—affords a spontaneity of interaction between users.

There are four physical NFC tag types. These are based on existing RFID standards; three of the tag types are based on the ISO-14443A standard, and the fourth is based on the ISO-18092 standard. This makes NFC tags at least partially compatible with many existing Mifare and FeliCa RFID systems. Although these older systems do not support NDEF, they can still identify those NFC Tags that are compatible with them. An RFID reader that can read Mifare Ultralight tags, for example, would still be able to read the UID of an NFC type 2 tag, even though it couldn't read any NDEF data encoded on the tag.

In its current state, there are some incompatibilities between some NFC devices and older Mifare Classic tags. Some NFC radios support Mifare Classic as an unofficial fifth tag type, while others don't. As NFC matures and more devices come on the market, these incompatibilities will hopefully come to affect users less. For now, however, it's best to check the compatibility between the readers and tags you plan to use in a given application.

CHAPTER 3

Getting Started with PhoneGap and the PhoneGap-NFC Library

PhoneGap is a development framework that allows you to build apps for iOS, Android, BlackBerry, Windows Phone 7 and 8, Symbian, and Bada all using (mostly) HTML5 and JavaScript. The folks at PhoneGap have created what is essentially a basic browser application, but with no interface. You implement the interface in HTML5 and JavaScript, then compile the app for your given platform. They've also developed a system for plug-ins so that developers can extend the basic browser application using native features of the various platforms. It's handy because it means you don't have to know the native application frameworks for the various mobile platforms in order to write applications that can run on them all.

For this book, you'll be using PhoneGap to write apps for Android and a plug-in that allows you to access the NFC hardware that's built into many Android phones. The code you write to listen for tags and to interact with the user will be written in JavaScript and the interface will be laid out in HTML5. The application shell is written in Java, as is the NFC plug-in, but you won't have to change any of the shell code for the projects in this book.

Why Android?

PhoneGap-NFC also supports Windows Phone 8, BlackBerry 7, and BlackBerry 10. We chose Android for this book because it has the largest market share and the most NFC phones, and most users of the PhoneGap-NFC plug-in have been Android users. If you have a Windows Phone or BlackBerry, you should be able to modify some of the examples to follow along. However, there are some limitations. Windows Phone 8 can read, write, and share NFC tags, but only has access to the NDEF data. It can't see tag type, tag UIDs, or any other tag metadata. BlackBerry 10 has similar restrictions: it can read and share NFC tags, but currently cannot write them.

PhoneGap-NFC for BlackBerry 7 supports more NFC features than BlackBerry 10, but you'll need to use older releases to make it work since PhoneGap 3.0 dropped support for BlackBerry 7.

You can use any text editor you want to write your code, but you'll use the Android software development kit (SDK) and its associated tools to compile your code and deploy it to your device. You'll also get to know the Cordova command-line interface (CLI), which is a set of tools for compiling PhoneGap applications using the Android SDK. You'll need a little familiarity with the command-line interface of your computer as well.

The instructions assume you're working in a command-line interface, such as the Terminal on any Linux machine or OS X. For Windows users at the Command Prompt, there are some slight differences that we'll call out.

Hello, World! Your First PhoneGap App

For the projects in this chapter, you need:

- A text editor (if you don't have a favorite text editor, we recommend Sublime Text 2, as it's a good cross-platform code editor)
- An NFC-enabled Android phone (see Table 2-1 for a list of usable phones)
- A few NFC tags (for best results across multiple devices, stick with the NFC Forum types; see "Device-to-Tag Type Matching" on page 19 for more on which tags work with which devices)
- The Android software development kit
- The Cordova CLI, your toolbox for PhoneGap ("Install Cordova CLI for PhoneGap" on page 28)
- Node.js and the Node Package Manager (npm); you'll use this to download the Cordova CLI (see "Install Node.js and npm" on page 27)

Setting Up the Development Environment

First you need to download the Android SDK from the Android Developer site. Android recommends that you download the Android Developers Toolkit (ADT) bundle, which includes a bundled version of the Eclipse development environment. We're not using Eclipse for this book, however, so you can just download the SDK tools for your platform, which should take less time to download. To do that, click "Use an Existing IDE" at the bottom of the "Download" page and you'll get a link to download just the SDK tools without Eclipse.

On OS X or Linux

Extract the Android SDK and put it somewhere on your computer where you know you can find it again. For example, if you put it in your Applications folder on OS X, the path to it would be */Applications/android-sdk-macosx*.

On Windows

You'll need to install a Java Development Kit (JDK) first by downloading it from the "Java SE" link on the Oracle website. After you install Java, you can run the Android SDK installer.

You'll need one more thing on Windows: Apache Ant, which manages the build process for Android programs. Download the Ant ZIP file from the Apache Ant website, extract the file to the *C:\* drive, and rename the top-level folder (such as *apache-ant-1.9.1-bin*) to *Ant*. If you did this correctly, you should find subdirectories such as *C:\Ant\bin* and *C:\Ant\lib*.

On Windows, the Android SDK installer will default to a hidden directory in your home directory: *C:\Users\Username\AppData\Local\Android\android-sdk*, which will make it hard to find later. We suggest you change this to *C:\Users\Username\Android\android-sdk*.

Install the Android Platform Tools

On OS X or Linux

Open up a Terminal window and change directories to the SDK directory (i.e., use `cd /Applications/android-sdk-macosx`). The SDK doesn't come with a version of the Android platform tools, so you need to install them by typing the following:

```
$ tools/android update sdk --no-ui
```

On Windows

If the Android SDK installer can't find your Java installation (a common problem on 64-bit Windows), quit the installer, and set your `JAVA_HOME` environment variable as directed later in this section. Try the installer again and it should find your Java installation.

You'll be given the opportunity to install the tools when the installer is finished. Leave the checkbox labeled "Start SDK Manager" checked and click "Finish." When the SDK manager appears, click "Install *n* Packages" (where *n* is whatever number of packages happen to be offered by default). If you accidentally unchecked the checkbox before clicking "Finish," open a Command Prompt, change directory to the SDK location (i.e., `cd Android\android-sdk`) and run the following command:

```
C:\Users\Username\Android\android-sdk>tools\android update sdk --no-ui
```

This will update the Android SDK and install the latest versions of the platform tools. You may be asked to accept some software licenses before you proceed. This will take a while, so you may want to take a minute to stretch and get some refreshment while it's installing.

Next you need to change the PATH variable for your environment to include the Android SDK tools. The tools directory covers most of the tools for compiling, uploading, and running your app on your device, and in platform tools, you'll find an important tool called adb that you'll use for debugging purposes later.

In the examples that follow, if you put the tools in a location other than the one shown, change the paths as needed. Also note that you are creating two paths: one for the *tools* subdirectory and one for *platform tools*.

On OS X or Linux

Change directories to your home directory and edit the *.bash_profile* or *.profile*. If the file doesn't exist, you can just create it. Add the following lines to that file:

```
export ANDROID_HOME=/Applications/android-sdk-macosx
export PATH=$PATH:$ANDROID_HOME/tools:$ANDROID_HOME/platform-tools
```

Change the *ANDROID_HOME* path if you stored the SDK somewhere other than the Applications directory, of course. Then save your profile file, and log out of the Terminal and log back in to reset the PATH variable.

On Windows

You have to find your way to the "System Properties Advanced" dialog box:

Windows XP

Open the Start Menu, then right-click on "My Computer" and choose "Properties," then go to the "Advanced" tab.

Windows 7 or Vista

Open the Start Menu and right-click on "Computer." Choose "Properties," then choose "Advanced System Settings" from the list on the left.

Windows 8.1

From the desktop, right-click or tap and hold on the Start Menu, then choose "System." Select "Advanced System Settings" from the list on the left.

Click the button at the bottom labeled "Environment Variables." Find the PATH entry under "User Variables" (*not* "System Variables") and click "Edit." If there is no PATH entry, click "New" and name the new variable PATH.

Append all three of the following to the end of the path with *no space* between them (the last two entries let you run Java and Ant from the command line):

```
;%ANDROID_HOME%\tools
;%ANDROID_HOME%\platform-tools
;%JAVA_HOME%\bin
;C:\Ant\bin
```

Be sure to include the `;` separator as shown (if you had to create a new `PATH` variable, you can skip the leading `;`). Don't change anything that's already there: you must add this to the end. If you make a mistake, click "Cancel" and try again. Stay out of the "System Variables." If you make a mistake in the system PATH setting, you can cause serious problems for Windows.

While you're on this screen, you must also set your `ANDROID_HOME` and `JAVA_HOME` environment variables:

1. Click "OK" to dismiss the PATH dialog, then click "New" under "User Variables."
2. Set the name to `JAVA_HOME` and the value to `C:\Program Files\Java\jdk1.7.0_25` (or whatever the installation directory of your JDK is).
3. Click "OK" to dismiss the PATH dialog, then click "New" again.
4. Set the name to `ANDROID_HOME` and the value to `C:\Users\%USERNAME%\Android\android-sdk`.

If you installed Java or the Android SDK in a different location, change the variables as necessary.

Finally, click "OK" until all the dialogs are gone.

Install Node.js and npm

Node.js is a platform for developing network applications in JavaScript. It also has a great system for installing packages (such as PhoneGap). Node's package manager (npm) is the fastest way to install the Cordova CLI for PhoneGap. You'll also see Node used in projects in Chapters 7 and 9.

To install Node.js:

Windows
    Download and install Node.js.

If you have any trouble running Node from the command line in Windows after you install it, you may need to log out and log in again for its PATH changes to take effect.

Linux
: Install Node.js using your package manager. If npm is in a separate package from Node.js, install it as well. For example, on Ubuntu, you'd run the command `sudo apt-get install nodejs npm`.

Mac OS X
: You can either use the Node.js installer on the website, or install it with a package manager such as Homebrew. If the installer doesn't add the npm module binary directory to your PATH, you may need to add it to your PATH by putting something like this in your *.bash_profile* or *.profile* and then open a new Terminal window:

```
export PATH=/usr/local/share/npm/bin:$PATH
```

You'll also need Git installed. It's installed by default on OS X and most Linux distributions. If not, install it with your package manager. You can get command-line Git for Windows, and the Git installer will ask if you want to install Git into the system PATH environment variable. We suggest that you do so, because this will let you run Git from the Command Prompt.

Install Cordova CLI for PhoneGap

Open a Terminal or Command Prompt and install the Cordova package (this gives you the PhoneGap framework).

On OS X or Linux
: Run the following command at a Terminal prompt:

```
$ npm install -g cordova
```

On Windows
: Run the following command in a Command Prompt window:

```
> npm install -g cordova
```

The `-g` option installs the Cordova command-line utility to a location in your PATH.

You can now create a PhoneGap project just by typing the following:

```
cordova create project-location package-name app-name
```

In the command, `project-location` is the path to your new Cordova Android project, `package-name` is the package name for the app you'll create, using reverse domain-style notation (e.g., com.example.myapp), and `app-name` is the name of your application.

You'll then need to `cd` into the project directory and add the Android platform:

```
cd /path/to/project-location
cordova platform add android
```

You'll see these steps frequently in the chapters that follow. For most of this book, we'll refer to the preceding steps simply by saying "use the `cordova create` command to create a project."

Now you're all ready to make your first PhoneGap project!

PhoneGap or Cordova?

What's the difference between PhoneGap and Cordova? PhoneGap was originally developed by Nitobi, which was later bought by Adobe. Before Nitobi was purchased, they donated the PhoneGap codebase to the Apache Software Foundation to ensure good open source stewardship of the code. As part of the transition from Nitobi to Apache, the project name was changed from PhoneGap to Cordova. PhoneGap is now Adobe's distribution of Cordova.

Historically, PhoneGap and Cordova, the open source project behind PhoneGap, were almost identical. This has changed with the release of 3.0. Now, the Cordova distribution provides the core functionality to embed a webview (browser) into a native application. Cordova is distributed without any of the plug-ins installed. This allows you to only install the functionality you need, simplifies your code base, and reduces the amount of code in your app.

PhoneGap is a distribution of Cordova, similar to how Safari and Chrome are based on WebKit. The PhoneGap distribution of Cordova comes with all of the core plug-ins installed. Additionally, the PhoneGap command line adds features like PhoneGap Build support. If you don't have a native SDK installed, the compiling of the project can be delegated to PhoneGap Build.

In this book, we use the terms PhoneGap and Cordova interchangeably. For code samples, we use the open source Cordova version.

Creating a PhoneGap Project

Change directories to whatever directory you like to keep projects in (perhaps your Documents directory) and use the `cordova create` command to create a project:

```
$ cordova create ~/Hello com.example.hello Hello
```

There is no output from `create` if it succeeds, but it will create a new directory. This command will create a directory in your working directory called *Hello*. Change directories to it and list the files:

```
$ cd ~/Hello
$ cordova platform add android
$ ls
```

> If Cordova requests that you install a specific Android target (other than the ones that were installed by default), type `android` to start the SDK manager, then install the SDK that Cordova requested. The version numbers of the SDK may not always match up (e.g., Cordova may ask for the Android 4.2 SDK, but the SDK manager only offers you 4.2.2), but the Android target that Cordova asks for (e.g., target 17) will match the API number listed in the SDK manager, as in Android 4.2.2 (API 17). After you install the Android target, try running the `cordova create` command again.

On Windows, the commands will be a little different. Open a Command Prompt (by default, it will place you in your home directory, usually *C:\Users\username*), but to be sure, you might want to change directory to it first with `cd /D %userprofile%` (in most configurations, the `%USERPROFILE%` environment variable contains your home directory). You can, of course, run these commands in another working directory:

```
> cd /D %userprofile%
> cordova create Hello com.example.hello Hello
> cd Hello
> cordova platform add android
> dir
```

On all three platforms, the directory will contain the following structure:

www
: The content files for the project: images, HTML, and so on. This is what you'll work with most.

platforms
: Support files for various platforms, such as Android.

plug-ins
: Any plug-ins you've installed.

merges
: Platform-specific web assets merged during the prepare phase.

The *platforms/android* directory contains the following:

AndroidManifest.xml
: The manifest file describing the app's structure, permissions, and so on.

ant.properties
: The compiler properties file.

assets
: A copy of the main *www* directory for the project.

bin
: Any binary files that are built in the compiling process.

build.xml
: The details needed by the compiler to build the project.

cordova
: A directory containing the various scripts that Cordova uses to automate the build process.

gen
: Files generated during the compile.

libs
: Any additional PhoneGap libraries needed, such as the PhoneGap-NFC library.

local.properties
: Properties of your development environment on your computer.

proguard-project.txt
: Properties for a tool for shrinking and obfuscating your code.

project.properties
: The project properties file.

res
: Resources used to build the project. You'll modify some files in this directory later.

src
: The Java source files for the app. You won't bother with these.

On Windows, you may need to install a USB driver for your device. When you updated the SDK earlier, you got USB drivers for many devices. However, even flagship devices like the Nexus 7 may require a separate driver installation. For example, you can download the Nexus 7 driver from ASUS. See Android's USB Drivers page for instructions on installing drivers that are included with the Android SDK.

You've got everything you need here to run a very basic app that does nothing but display a splash screen. To compile and install it on your device, connect the device over USB and run the following command if you are on OS X or Linux:

```
$ cordova run
```

If you're on Windows, run this command:

```
> cordova run
```

When everything's ready, you'll be running your app directly on your Android phone or tablet.

Can't find developer options on Android version 4.2 or later? Open "Settings" and scroll down to "About Device" (it's "About Tablet" on the Nexus 7). Tap the "Build" number seven times. This will enable developer options. Then return to the previous screen to find "Developer Options" and turn it on. In particular, you want to enable "Stay Awake" while charging and "USB Debugging." If you have a password set for your device, you may want to disable it while you're coding, as it removes the need to unlock your device every time you upload a new version of your app.

Finally, open the phone and launch the Hello Application. You should see a screen like that in Figure 3-1. Once you've got the basic "Hello, World!" application running, try modifying the *index.html* to make it more fancy. It lives in the *www* directory. Add buttons, images, and all the other things you'd add to an HTML page. Then save your files, run it again, and see how it changes on your phone.

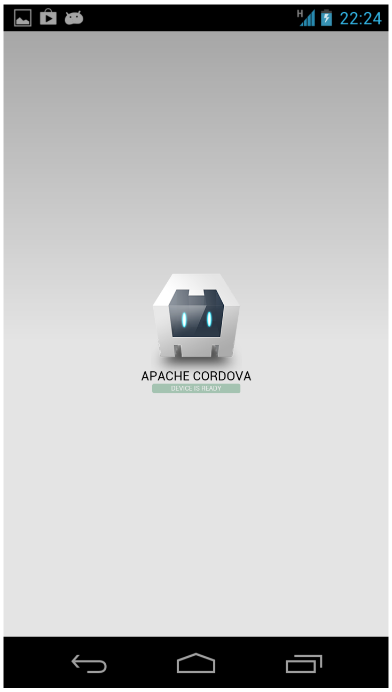


Figure 3-1. The PhoneGap HelloWorld app

The Important Files

There are two files you'll work with frequently in your PhoneGap projects: the *index.html* in the *www* directory, and the *index.js* file in the *www/js* directory. You can add other HTML and JavaScript files, but by convention, these will be the root of your application. These are the files you'll modify to make your app do what you want.

The *index.html* page will include the graphic layout of your application, just as it would for any website. The *index.js* contains event handlers, like `onDeviceReady()`, `initialize()`, and so forth. Later on you'll add some functions to it, like `onNfc()` for when a tag is read.

You can always get an unmodified version of the default file structure by creating a new project.

There are a few other JavaScript files that affect your application that you'll likely never see: the *cordova.js* file, the *cordova_plugins.js* file, and its companion, the *cordova_plugins.json* file. The *cordova.js* file is the main PhoneGap library. The *cordova_plugins.js* and *cordova_plugins.json* files initialize the plug-ins you install, giving you access to the PhoneGap-NFC library and the NFC hardware. These show up in *platforms/android/assets/www* when you install plug-ins and run your app.

A Word on JavaScript Idioms

The default coding idiom, or pattern of programming, for PhoneGap may look strange to you if you're still new to JavaScript. In many JavaScript programs, you'll see functions initialized like this:

```
function toggleCompass() {

}
function init() {

}
```

If you've programmed in C or Java, this pattern probably looks familiar to you. These functions are global to your application, and can be called from anywhere like this:

```
toggleCompass()
```

In the default Cordova *index.js* file, on the other hand, you'll see that all functions are created inside one giant variable called `app.` In JavaScript, variables can contain functions, and you can pass functions as parameters of functions. Since they're elements of the `app` object, they're written as an *object literal*, like so:

```
var app = {
   initialize: function() {

   },
   bindEvents: function() {

   },

   onDeviceReady: function() {

   }
}
```

An object literal is a pair of curly braces surrounding name/value pairs, which are separated by commas inside the braces. If you've seen *JavaScript Object Notation* (JSON), you've seen this before. JSON is the syntax for writing object literals. These are still functions, just in a different notation. To call these functions, which are local to the `app` object, you do it like so:

```
    app.initialize();
    app.bindEvents();
    app.onDeviceReady();
```

Writing all your functions inside the `app` object means that you know they're all local and you won't accidentally overwrite functions from other JavaScript libraries with the same names you may choose to use in more complex apps.

For more on JavaScript, check out Douglas Crockford's *JavaScript: The Good Parts*. It does a nice job of describing how JavaScript is structured and how it works.

A Simple Locator App

Since the "Hello, World" template is such a handy reference, keep it intact and make a new project for your first custom app. This project will contain the bare minimum you need in your HTML and JavaScript files to get things running. To make it interesting, you'll make an app that starts and stops tracking your latitude and longitude on a click.

In the following examples, OS X and Linux (indicated by a leading `$`) and Windows (indicated by a leading `>`) variants of the commands are shown one after another. On Windows, we're using %userprofile%, which points to your home directory in most Windows installations, but you can use another location if you wish.

In future chapters, unless the commands are radically different, we'll only show the OS X and Linux version, and Windows users should assume the commands are differentiated as shown here.

Make a new project called Locator just as you did before. Use `cordova create` again:

```
$ cordova create ~/Locator com.example.locator Locator ❶
```

❶ Windows users should type `%userprofile%\Locator` instead of `~/Locator`.

Then add the Android platform and the geolocation plug-in to the project. Plug-ins for PhoneGap/Cordova extend the framework's functionality. Plug-in authors register their plug-ins' URLs with the Cordova project so they can be added to the plug-in database. Many plug-ins add access to a device's hardware, like this one, which gives you access to a device's geolocation system. The NFC plug-in, which you'll see shortly, gives you access to the NFC radio. You'll see how plug-ins fit into the file structure a bit later, but for now, here's how to install the geolocation plug-in:

```
$ cd ~/Locator ❶
$ cordova platform add android
$ cordova plugin add \  ❷
    https://git-wip-us.apache.org/repos/asf/cordova-plugin-geolocation.git
```

❶ Windows users should type `/d %userprofile%\Locator` instead of `~/NfcReaderLocator`.

❷ \ is the line continuation character in Linux, OS X, and other POSIX systems.

Now change directories to the project's *www* directory. Edit the *index.html* file and delete everything. The following is a minimal HTML file that includes only the basics you'll use for most apps. There's a main div element called `app` where all the action will take place. You'll see it and its subelements modified quite a bit in the *index.js* file. At the end of the file, include the *cordova.js* script and your application's *index.js* script. Finally, call the `initialize()` function from *index.js* to run the app:

```
<!DOCTYPE html>

<html>
   <head>
      <title>Locator</title>
   </head>
   <body style="font-size: 1.4em;">
      <div class="app">
         <div id="messageDiv"></div>
      </div>
      <script type="text/javascript" src="cordova.js"></script>
      <script type="text/javascript" src="js/index.js"></script>
      <script type="text/javascript">
       app.initialize();
     </script>
   </body>
</html>
```

Edit the *index.js* file and delete everything. You'll replace it with the following script. All the functions and variables in this script are local to an object called `app`, by convention. Start out by initializing a variable called `app`:

```
var app = {

};
```

Next, add a few functions you need for initialization, to set up listeners for startup events, button touches, and the like. Replace the code you just added with this:

```
var app = {
/*
   Application constructor
*/
   initialize: function() {
      this.bindEvents();
      console.log("Starting Locator app"); ❶
   },
/*
   bind any events that are required on startup to listeners:
*/
   bindEvents: function() {
      document.addEventListener('deviceready', this.onDeviceReady, false); ❷
   },

/*
   this runs when the device is ready for user interaction:
*/
   onDeviceReady: function() {
      app.display("Locating...");
      app.watchLocation();
   },
```

❶ `console` is defined by your browser. Every modern browser features a JavaScript console viewer in its interface.

❷ `document` is defined by your browser and the HTML document that you're viewing.

Next, add a function to get the device's location. This function has two callbacks, one for success and one for failure. If it succeeds, it displays the device's coordinates using a function called `display()`. If it fails, it displays a failure message:

```
/*
   Displays the current position in the message div:
*/
   watchLocation: function() {
       // onSuccess Callback
       // This method accepts a `Position` object, which contains
       // the current GPS coordinates
       function onSuccess(position) {
           app.clear();
           app.display('Latitude: '  + position.coords.latitude);
           app.display('Longitude: ' + position.coords.longitude);
           app.display(new Date().toString());
```

```
            }

            // onError Callback receives a PositionError object:
            //
            function onError(error) {
                app.display(error.message);
            }

            // Options: throw an error if no update is received every 30 seconds.
            //
            var watchId = navigator.geolocation.watchPosition(onSuccess, onError, {
             timeout: 30000,
             enableHighAccuracy: true
             });
        },
```

Finally, you need to connect these functions to the user interface. The functions that follow, `display()` and `clear()`, write to an HTML `div` element in *index.html*. You'll see these same two functions in many apps in this book:

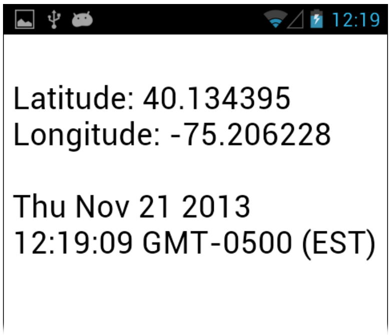
```
        /*
           appends @message to the message div:
        */
        display: function(message) {
           var label = document.createTextNode(message),
              lineBreak = document.createElement("br");
           messageDiv.appendChild(lineBreak);          // add a line break
           messageDiv.appendChild(label);              // add the text
        },
        /*
           clears the message div:
        */
        clear: function() {
            messageDiv.innerHTML = "";
        }
    };      // end of app
```

The full source listing for this application can be found on GitHub.

Save the files you edited, then `cd` to the project root, and run the app on your phone with the following commands:

```
    $ cd Locator
    $ cordova run

    > cd /d %userprofile%\Locator
    > cordova run
```

When you run the app on the phone, you'll get a screen as shown in Figure 3-2.


Latitude: 40.134395
Longitude: -75.206228

Thu Nov 21 2013
12:19:09 GMT-0500 (EST)

Figure 3-2. The Locator app showing location

You can modify the HTML and JavaScript just as you would for any other HTML-, JavaScript-, or CSS-based application on the Web. So feel free to experiment. Once you understand the basics of making a PhoneGap app, you're ready to move on.

Ways to Debug

There are a few useful tools for debugging while you're working on PhoneGap apps. Use whichever of these you find most helpful.

First, you might want to install jshint to check your code before running it. To install it, type:

```
$ npm install -g jshint
```

Then to run it, type:

```
$ jshint /path/to/js/file
```

When you run your code through jshint, it will pick up errors and formatting mistakes before you even run it.

When your device is still tethered to your computer, you can also get a log output when your app runs using `adb logcat`. After you run your app with `cordova run`, type:

```
$ adb logcat
```

This will give you the log of everything that happens on your Android phone. It's pretty detailed, and includes more than just your JavaScript *console.log* messages. It can be useful, but it can also be overwhelming. For example, here's the output from the launch of the previous app:

```
I/ActivityManager(  393): START u0 {act=android.intent.action.MAIN
        cat=[android.intent.category.LAUNCHER] flg=0x10200000
        cmp=com.example.locator/.Locator} from pid 590
D/dalvikvm( 2883): Late-enabling CheckJNI
I/ActivityManager(  393): Start proc com.example.locator for activity
```

```
        com.example.locator/.Locator: pid=2883 uid=10080
        gids={50080, 1006, 3003, 1015, 1028}

... twenty or so lines cut for brevity's sake ...

D/SoftKeyboardDetect( 2883): Ignore this event
I/ActivityManager(  393): Displayed com.example.locator/.Locator: +475ms
D/CordovaLog( 2883): file:///android_asset/www/js/index.js: Line 7 :
        Starting Locator app
I/Web Console( 2883): Starting Locator app at file:///android_asset/
        www/js/index.js:7
```

If you want to filter out all but the web console messages, run it through `grep` (OS X or Linux) or `findstr` (Windows) like so:

```
$ adb logcat | grep "Web Console"
```

```
> adb logcat | findstr /C:"Web Console"
```

When you do this, any of the `console.log()` statements in your JavaScript will show up in the console. It'll focus on things that you need to know most of the time. The output is stripped down to:

```
I/Web Console( 2335): Starting Locator app
        at file:///android_asset/www/js/index.js:7
```

You can also filter by any of the other message types this way. To quit it, press ctrl+C. When you disconnect your device from your computer, it will stop automatically.

Some programmers prefer to keep it running in a separate Terminal or Command Prompt window so they can compile and run without having to start and stop it. You can differentiate between runs of your app by hitting Enter several times in that window to create a break. Others prefer the stop/start method because it gives you a clearer record for each time you test your app.

You can also use the Android Debug Monitor to watch everything going on. This is a graphic user interface that lives in the *tools* subdirectory of your Android SDK. Because you added that to your path earlier on, you can simply type the following to launch it:

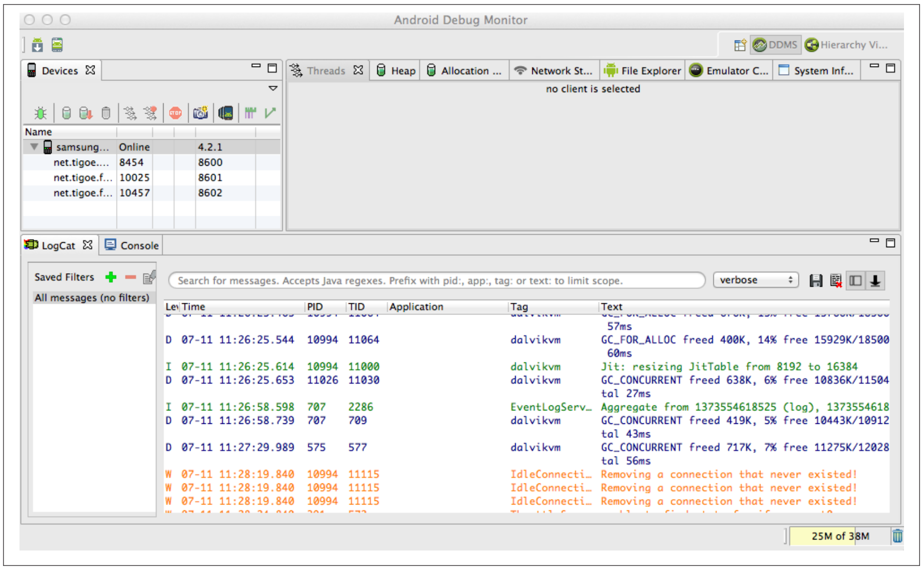
Linux or OS X:

```
$ monitor &
```

Windows:

```
> start monitor
```

The ampersand will cause monitor to run in the background and return you to the Command Prompt after launching. If you are using Windows, the batch file starts the monitor in the background. You can filter the logcat messages using the "Web Console" tag, or any other log tag you want to look for. You can also see the status of the other logs as well. There's also a performance monitor, a file explorer, and more. Figure 3-3 shows the interface.



```
Le Time                PID    TID    Application   Tag               Text
                                                                      57ms
D  07-11 11:26:25.544  10994  11064                dalvikvm          GC_FOR_ALLOC freed 400K, 14% free 15929K/18500
                                                                      60ms
I  07-11 11:26:25.614  10994  11000                dalvikvm          Jit: resizing JitTable from 8192 to 16384
D  07-11 11:26:25.653  11026  11030                dalvikvm          GC_CONCURRENT freed 638K, 6% free 10836K/11504
                                                                     tal 27ms
I  07-11 11:26:58.598  707    2286                 EventLogServ...   Aggregate from 1373554618525 (log), 1373554618
D  07-11 11:26:58.739  707    709                  dalvikvm          GC_CONCURRENT freed 419K, 5% free 10443K/10912
                                                                     tal 43ms
D  07-11 11:27:29.989  575    577                  dalvikvm          GC_CONCURRENT freed 717K, 7% free 11275K/12028
                                                                     tal 56ms
W  07-11 11:28:19.840  10994  11115                IdleConnecti...   Removing a connection that never existed!
W  07-11 11:28:19.840  10994  11115                IdleConnecti...   Removing a connection that never existed!
W  07-11 11:28:19.840  10994  11115                IdleConnecti...   Removing a connection that never existed!
```

Figure 3-3. The Android Debug Monitor

If your device is connected to the Internet, you can also find a JavaScript console at *http://debug.phonegap.com*. Just put in a unique name like "kramnitz" in the field entry on that page, then include the script link in your *index.html* file. The next time you run your app, you'll see your JavaScript console at *http://debug.phonegap.com/client/#kramnitz*. It's handy and familiar, but it does require a network connection.

The advantage of *debug.phonegap.com* is also its disadvantage: you only see your JavaScript errors there. Sometimes you catch bugs from Android's messages using `adb logcat` that you'd otherwise miss if you were only looking for JavaScript errors. You may also find that *debug.phonegap.com* runs a bit slower when there's a heavy load of other programmers on it.

PhoneGap Meets NFC: NFC Reader

Now that you've got a handle on PhoneGap, it's time to extend it to include the NFC plug-in. For this you'll need the same software as the first app, the PhoneGap-NFC plug-in, and a few NFC tags as well. The first NFC app you create will read just the unique ID of any tag it sees.

The same structure will be used for all PhoneGap apps, so keep this app handy to use as a template. Create a new app called `NfcReader`:

```
$ cordova create ~/NfcReader com.example.nfcreader NfcReader ❶
$ cd ~/NfcReader ❷
$ cordova platform add android
```

❶ Windows users should type `%userprofile%\NfcReader` instead of `~/NfcReader`.

❷ Windows users should type `/d %userprofile%\NfcReader` instead of `~/NfcReader`.

Installing the NFC Plug-In

There are a number of changes needed to add NFC support to your project. These are all automated by the command:

```
$ cordova plugin add /path/to/plugin
```

The one you'll use commonly throughout the book is:

```
$ cordova plugin add https://github.com/chariotsolutions/phonegap-nfc.git
```

So you understand what's going on under the hood, here's what's installed:

- The plug-in's files live in your project's *plugins/* subdirectory (in this case, it's called *com.chariotsolutions.nfc.plugin*). It also gets installed in *platforms/android/assets/www/plugins/*.
- Your project's *AndroidManifest.xml* file and the resource config files also need modification so your app can use the NFC reader hardware on your device. This file lives in your project's *platforms/android/* directory.
- The file *platforms/android/res/xml/config.xml* gets modified as well. Here are the modifications to those two files:

  ```
  <feature name="NfcPlugin">
     <param name="android-package"
      value="com.chariotsolutions.nfc.plugin.NfcPlugin" />
  </feature>
  ```

- The *AndroidManifest.xml* file gets the NFC permission added at the end of the other permissions tags:

  ```
  <uses-permission android:name="android.permission.NFC" />
  <uses-feature android:name="android.hardware.nfc"
                android:required="false" />
  ```

Since that's a lot to remember, it's nice that the one-line plug-in `add` command does it all for you.

You've already created your project using the `create` command, as directed earlier. Next, change directories to the root of your project and add the plug-in as described at the beginning of this section like so:

```
$ cd ~/NfcReader
$ cordova plugin add https://github.com/chariotsolutions/phonegap-nfc.git
```

```
> cd /d %userprofile%\NfcReader
> cordova plugin add https://github.com/chariotsolutions/phonegap-nfc.git
```

This installs the NFC plug-in from its online repository in the current working directory.

Writing the NFC Reader App

Now that you've got all the assets in place, it's time to make the application itself. You'll have two files you need to edit for this project:

- *index.html* in the *www* directory
- *index.js* in the *www/js* directory

This app will listen for NFC-compatible RFID tags and print their IDs on the screen.

Start with *index.html* as before. Here's a bare-bones page, much like the Locator app's index page. The changes are that you're adding a call to the Cordova script at the end of the body and you're changing the elements of the app div in the body. The latter is where tag IDs will be displayed.

Open *index.html* and replace it with the following code, then save the changes:

```
<!DOCTYPE html>

<html>
   <head>
      <title>NFC tag ID reader</title>
   </head>
   <body style="font-size: 1.5em">
      <div class="app">
         <div id="messageDiv"></div>
      </div>
      <script type="text/javascript" src="cordova.js"></script>
      <script type="text/javascript" src="js/index.js"></script>
      <script type="text/javascript">
       app.initialize();
     </script>
   </body>
</html>
```

The *index.js* file will have an event handler to listen for NFC tags and rewrite the message div when it gets a tag. As before, start by initializing the app variable, and adding an `initialize()` function and a `bindEvents()` function:

```
var app = {
/*
   Application constructor
*/
```

```
    initialize: function() {
        this.bindEvents();
        console.log("Starting NFC Reader app");
    },

  /*
    bind any events that are required on startup to listeners:
  */
    bindEvents: function() {
        document.addEventListener('deviceready', this.onDeviceReady, false);
    },
```

The `onDeviceReady()` function is a bit more complex this time. You need to add a listener for when tags are discovered by the NFC reader. The `nfc.addTagDiscoveredListener()` function tells the NFC plug-in to notify your app when any NFC tag is read. The first argument is the event handler that is called when a tag is scanned. The second and third arguments are the success and failure callbacks for the plug-in initialization. When it succeeds, it will call the `onNfc()` function. When the listener is initialized, it'll let you know, and when it fails, it'll give you failure messages as well.

The `addTagDiscoveredListener()` handler is one of a few different listeners you can use from the NFC library. It's the most generic. It doesn't care what's on the tag, it just responds whenever it sees a compatible tag:

```
  /*
    this runs when the device is ready for user interaction:
  */
    onDeviceReady: function() {

        nfc.addTagDiscoveredListener(
            app.onNfc,             // tag successfully scanned
            function (status) {    // listener successfully initialized
                app.display("Tap a tag to read its id number.");
            },
            function (error) {     // listener fails to initialize
                app.display("NFC reader failed to initialize " +
                    JSON.stringify(error));
            }
        );
    },
```

The `onNfc()` function takes the tag read in an NFC event and prints it to the screen using `display()`, just as you saw in the Locator app. These are the last functions in the app:

```
  /*
    displays tag ID from @nfcEvent in message div:
  */
    onNfc: function(nfcEvent) {
        var tag = nfcEvent.tag;
        app.display("Read tag: " + nfc.bytesToHexString(tag.id));
    },

 /*
    appends @message to the message div:
 */
    display: function(message) {
       var label = document.createTextNode(message),
          lineBreak = document.createElement("br");
       messageDiv.appendChild(lineBreak);         // add a line break
       messageDiv.appendChild(label);             // add the text
    },
 /*
    clears the message div:
 */
    clear: function() {
        messageDiv.innerHTML = "";
    }
 };     // end of app
```

The full source listing for this application can be found on GitHub.

That's all the changes. Save both files, then change directories to the root of the project if you're not already there, and compile and install the application:

```
$ cd NfcReader
$ cordova run
```

```
> cd /d %userprofile%\NfcReader
> cordova run
```

This launches the NFC Reader application. When it's up and running, bring a tag close to the phone, and you should get a message saying "Read tag" followed by the UID, in hexadecimal notation, as shown in Figure 3-4. Try with another tag and you'll see another UID.

Project Repository Housekeeping

Once you've got a project working, if you're going to save it in a Git repository or any other version control system, you may not want to store the build files, binaries, and so forth. It's a good idea to delete or `.gitignore` the file *platforms/android/local.properties* in your project directory so it's not uploaded to the repository. If you've got the PATH variables set in your login profile as shown in "Install Node.js and npm" on page 27, you'll never need the *local.properties* file. If you don't, then when you're updating the project from a remote repository, you'll need to:

```
$ cd projectDirectory
$ android update project -p platforms/android/
```

Here's a list of everything you can safely omit from your repository. If you're using Git, you can simply `.gitignore` these:

```
    platforms/android/local.properties
    platforms/android/bin
    platforms/android/gen
    platforms/android/assets/www
```

Tap a tag to read its id number.
Read tag: fa4360e6

Figure 3-4. The NFC Reader app reading a tag

This will work with any RFID tag that's compatible with your NFC reader. That includes all tags compatible with the ISO-14443A format, including Philips and NXP Mifare tags, Sony FeliCa tags, NXP DESFire. You're not yet reading or writing data to the tag; you're just reading the tag's unique ID. For this application, compatible tags are not as much of an issue as when you start to write NDEF records to the tags. You'll find that this app can read Mifare Classic tag IDs even on those devices that are supposedly incompatible with Mifare Classic.

You've proven that your device's reader works, and can be programmed to read tags. In the next chapter, you'll learn how to read from and write to tags using the NFC Data Exchange Format, NDEF.

Troubleshooting

If your app doesn't read any of your tags, here are a few things to check:

Is your device NFC-enabled?
: To check, go to "Settings" on your device and tap "More…" If you don't see NFC there, you're out of luck. If you do see it, make sure it's enabled. When you scan a tag, you should get an acknowledgment melody. A successful read melody ends on a high note, and a failed read ends on a low note. The PhoneGap-NFC plug-in will report in the error if NFC is disabled or if NFC is not available. This app prints that info to the screen. Others may not.

Are you using compatible tags?
: If you get no acknowledgment melody when you scan a tag, make sure it's a compatible tag. If it's not, your reader won't read it.

Did you make good contact?
: The NFC reader on most devices is only part of the back of the device, usually near the upper part. If you don't get a read, try moving the tag around the back to find out where the reader's antenna is. It sometimes takes a second or so to get a good read.

Are your tags damaged?
: Ripped, wrinkled tags won't work, nor will tags placed perpendicular to the surface of the device. Make sure your tags are in good condition and are flat against the device.

Conclusion

The steps you followed in this chapter will be used in all of the PhoneGap-NFC projects that follow in this book, so here's a quick summary to use for future reference:

- Initialize a generic PhoneGap app with the `cordova create` command
- Add the platform using `cordova platform`
- Add the PhoneGap-NFC assets using `cordova plugin`

Your app needs, at minimum, the following files:

- *index.html*, in the *www* directory
- *index.js*, in the *www/js* directory

Your *index.html* file needs to include the following JavaScript scripts:

- *cordova.js*

- *index.js*

Your main JavaScript file should include the following elements from the PhoneGap-NFC library:

- An NFC listener handler
- A function to respond to NFC events

If you've got all of those elements, you're ready to go.

By now, you should be familiar with the structure of a PhoneGap app, and with the steps necessary to add the PhoneGap-NFC plug-in to an app. You'll see this same structure replicated in the rest of the projects in this book. What you've seen so far, though, is really just RFID—there's been no real NFC action yet. In the next chapter, you'll learn about the data exchange format that defines all NFC transactions.

CHAPTER 4

Introducing NDEF

In order to understand NFC, you need to know about the NFC Data Exchange Format (NDEF), which is the lingua franca for NFC devices and tags. In this chapter, you'll learn about the structure of NDEF and the records it carries. You'll also write a couple of apps that read and write NDEF-formatted messages.

NDEF Structure

NDEF is a binary format structured in *messages*, each of which can contain several *records*, as shown in Figure 4-1. Each record is made up of a *header*, which contains metadata about the record, such as the record type, length, and so forth, and the *payload*, which contains the content of the message. Think of an NDEF message like a paragraph, and records like the sentences within it. A well-formed paragraph is made up of sentences pertaining to one topic. Similarly, it's good practice to use one NDEF message made up of several records to describe one subject, say, an address book entry.

NFC transactions are generally short. Each exchange generally consists of only one message, and each tag carries just one message. Keep in mind the physical circumstances of an NFC exchange: you tap your device to another device or tag, and the whole exchange happens while you're in contact with the other device or tag. You don't want to send a whole novel in a single exchange, so think of your NDEF messages as paragraph-length, not book-length. You'll see a workaround to this for sending large files in one of the final chapters of this book, but for now, consider one NFC exchange as one NDEF message, and think of one NDEF message as one or more short records.

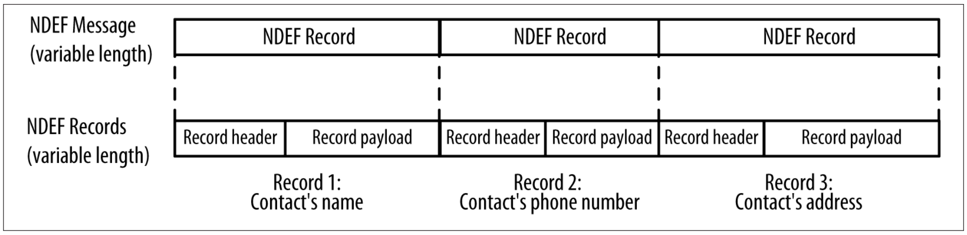


Figure 4-1. The structure of an NDEF message made up of several records; this is a typical example—an address book entry with three records (name, phone number, address)

An NDEF record contains a payload of data and metadata describing how to interpret the payload. Each record's payload can be one of several different *data types*. The header for each record contains metadata describing the record and its place in the message, followed by its type and ID. After the header comes the payload. Figure 4-2 gives you the full picture of the bits and bytes of an NDEF record.

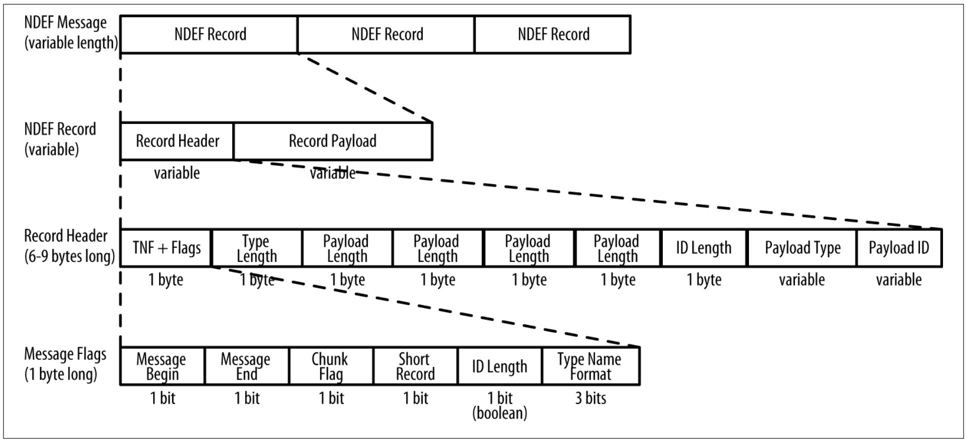


Figure 4-2. NDEF message structure, with details about the bytes of the header

As you can see from Figure 4-2, an NDEF record consists of a type name format (TNF), payload type, payload identifier, and the payload. The payload is the most important part of an NDEF record; it's the content that you're transmitting. The TNF tells you how to interpret the payload type. The payload type is an NFC-specific type, MIME media-type, or URI that tells you how to interpret the payload. Another way to think about this is that the TNF is the metadata about the payload type, and the payload type is the metadata about the payload. The payload identifier is optional and allows multiple payloads to be associated or cross referenced.

Type Name Format

NDEF records begin with a type name format. The TNF indicates the structure of the value of the type field. The TNF tells you how to interpret the type field. There are seven possible TNF values:

0 Empty
: Empty record that has no type or payload.

1 Well-Known
: One of several pre-defined types laid out in the NFC Forum RTD specification.

2 MIME media-type
: An Internet media type as defined in RFC 2046.

3 Absolute URI
: A URI as defined in RFC 3986.

4 External
: A user-defined value, based on rules in the NFC Forum Record Type Definition specification.

5 Unknown
: Type is unknown. Type length must be 0.

6 Unchanged
: Only for middle and terminating records of chunked payloads. Type length must be 0.

7 Reserved
: Reserved by the NFC Forum for future use.

For many applications, you'll probably use TNF 01 (Well-Known) or TNF 02 (MIME media-type) for various Internet media. You'll see the TNF 04 (External) frequently as well, since Android uses an External type called an Android Application Record to trigger apps to open.

Payload Type

The Payload type, also known as the Record type, describes the content of the payload more specifically. The TNF defines the format of the payload type. The type can be an NDEF-specified type, MIME type, URI, or External type. The NDEF Record Type Definition (RTD) specification describes Well-Known Record types and sets the rules for creating External types. The MIME RFC and the URI RFC set the rules for the other types.

For example, a record of TNF 01 (Well-Known) could have a Record type of "T" for a text message, "U" for a URI message, or "Sp" if the payload is a Smart Poster. A record of TNF 02 (MIME media-type) might be one of several different Record types, among them "text/html," "text/json," and "image/gif." A record of TNF 03 (Absolute URI) would have a literal URI, *http://schemas.xmlsoap.org/soap/envelope/*, as the type. A record of TNF 04 (External) might also have one of several different Record types, the most common of which you'll see in this book is "android.com:pkg."

For more detail, consult the NDEF Specification document on formats for NFC Forum Record Type Definitions. You can find a list of common NFC RTD specifications in Appendix A as well.

NDEF messages can contain multiple Payload types, but by convention, the type of the first record determines how the whole message is processed. For example, Android's intent filtering for NDEF messages only looks at the first record.

URIs in NDEF Messaging

You'll hear the terms URI, URL, and URN frequently in this book. A URI, or uniform resource identifier, is a string of characters that identifies a web resource. A URN, or uniform resource name, names the URI. A URL, or uniform resource locator, also tells you the type of transport protocol needed to get the resource. If the URN is your name, then the URI is your address, and a URL is telling someone to take the bus to get to your address. Or in web terms:

URN
: mySpecialApp

URI
: net.tigoe.mySpecialApp (in reverse domain format)

URL
: *http://tigoe.net/mySpecialApp*

The type name format "Absolute URI" is a bit misleading. The Absolute URI TNF means that the *Record type*, not the *payload*, is a URI. The URI in the type field describes the payload, similar to the way a MIME type describes the payload for TNF 02. For example: Windows and Windows Phone use TNF 03 (Absolute URI) for LaunchApp records using the URI "windows.com/LaunchApp." LaunchApp records prompt the user to launch an app, much like Android uses Android Application Records to launch apps. If the app is not installed, the user is prompted to download it from the store.

Android breaks the NDEF specification a bit with Absolute URI records. Although the NDEF specification says the type describes the payload, Android devices handle TNF 03 (Absolute URI) records by opening the browser with the URI in the type field; basically, Android treats the URI in the type field as if it were the payload. BlackBerry and Windows Phone do not open the browser when an Absolute URI tag is scanned.

If you want to send a URI or URL as a *payload*, then you shouldn't use TNF 03 (Absolute URI). You should encode them as TNF 01 (Well-Known) with NFC RTD "U" (URI). The NDEF specification provides a URI Record Type Definition specification with URI Identifier Codes for encoding URIs more efficiently. For example, 0x01 is the code for *http://www*. 0x02 is the code for *https://www*. In "Writing Different Record Types" on page 66 you'll see an example in which you add a URL to a payload, then add a single byte with the value 0x01 to add the *http://www*. You'll see these in more detail in the application that follows, and you can see them all in Appendix A.

You can also encode URIs onto a TNF 01 (Well-Known) with NFC RTD "Sp" (Smart Poster). Using Smart Poster records allows additional information to accompany the URI, such as text descriptions in multiple languages, icons, and optional processing instructions.

Payload Identifier

The payload identifier, which is an optional field, should be a valid URI. It can be a relative URI, so even "foo" would work. It's used to let your applications identify the payload within the record by ID or to allow other payloads to reference other payloads. It's up to you to decide on the payload ID, but you also don't have to include it.

Payload

The payload is your content. It can be anything you want that fits in a stream of bytes. A properly constructed NDEF library doesn't care what's in the payload, it just passes it on. You can encrypt your payload, you can send plain text, you can send a binary blob, or anything else you can think of. It's up to the sending application and the receiving application to agree on what the payload means and how it's formatted.

Figure 4-2 provides some more details about the binary format of the record header's first byte. You will hopefully never need to use this information, since any well-written NDEF library should handle all of this for you. If you're using the software libraries in this book, this is true. The finer details of the binary formatting will be taken care of for you, and can safely skip ahead to "NDEF in Practice" on page 56. If you want to understand more of the structure of messages and records, read on.

Record Layout

The first five bits of the NDEF record are flags that tell how to process the record and information about the record's location in the message.

The bit flags in the first byte of the record header are as follows:

MB (Message Begin)
  True when this is the first record in a message.

ME (Message End)
: True when this is the last record in a message.

CF (Chunk Flag)
: True when this record is chunked.

SR (Short Record)
: True if the short record format is used for payload length.

IL (ID Length is present)
: True if the ID Length field is present.

Note that the IL bit is not the length of the ID field, it just indicates the presence of the length field. If the IL bit is 0, there is no ID length field or ID field.

The Message Begin and Message End bit flags are used for processing the records within a message. Since an NDEF message is just a collection of one or more NDEF records, there is no binary format for an NDEF message. The MB and ME flags let you determine when the message begins and ends. The first record in a message will have its MB flag set to true. The middle records will have both flags set to false. The end record in a message will have the ME set to true. A message with one record will have both the Message Begin and Message End bits set true.

Since there are only eight possible type name formats, you only need 3 bits to store it. The TNF is stored in the last three bits of the Message Flags byte.

Record Header

NDEF records are variable length data structures. The record header contains the information required to read the data.

An NDEF record begins with the TNF byte, which includes the bit flags. After the TNF, the NDEF record header includes the type length. The type length is a one byte field that specifies the length of the payload type in bytes. The type length is required, but may be zero.

The payload length comes next. The Short Record (SR) bit flag in the first byte of the record header determines the length of the payload record. If SR is true, the payload length is one byte, otherwise it's four bytes. Payload length is required, but may be zero.

If the ID length field is present and the (IL) flag is true, the next byte in the header is the ID length.

The record type field is a variable length field after ID length field (or after the payload length field if the IL flag is false). The type length field determines how many bytes should be read.

If there is a record ID, it comes after the type. This variable length field is determined by the ID length byte.

This is the end of the header. The payload comes next.

How Big Can an NDEF Message Be?

NDEF record payloads are limited in size to 2^{32}–1 bytes long, which is why the payload length field of the header is four bytes (or 2^{32} bits). However, records can be chained together in a message to form longer payloads. In theory, there is no limit to the length of an NDEF message. In practice, the capabilities of your devices and tags define your limits. If you're exchanging peer-to-peer messages between devices and no tags are involved, your NDEF messages are limited only by the computational capacity of your devices, and the patience of the person holding the two devices together. If you're communicating between a device and a tag, however, your messages are limited by the tag's memory capacity.

When you're using NFC tags, the size limits of your records are well below the 2^{32}–1 byte limit. NFC tag types are based on a few different RFID tag standards. Most NFC tag types are based on ISO-14443A standard. These vary from 96 bytes of memory, expandable to 4K depending on the tag type. The Philips/NXP Mifare family of tags are compatible with NFC, including the Mifare Ultralights, Mifare Classic 1K and 4K tags, and the Classic Mini. There is one type of NFC tag based on the Japanese Industrial Standard (JIS) X 6319-4. These have up to 1MB of memory. Sony FeliCa tags are typical of this type. For more details on the tag type specifications, see the specifications section of the NFC Forum website.

Generally, NFC exchanges are short. A person holds her device to a tag or another device, a brief exchange happens, and she moves on. It's not a protocol designed for long exchanges, because the devices need to be held literally in contact with each other. When you send large messages, the user has to hold the device in place for as long as the message takes to transfer. This can get tedious, so people generally prefer to use NFC to exchange capabilities between devices, then switch, or handover, to WiFi or Bluetooth to exchange data or media files.

Here's a typical example of where NFC and WiFi might work in tandem: imagine you've got an NFC-enabled home music player that can sync tracks with your smartphone or tablet via WiFi from a central home media server. You're listening to an album on the home stereo, but you have to leave for work. You tap your phone to the stereo, and the stereo tells the phone via NFC what track is playing and what time it's up to in the song. Your phone then checks to see if it's got the song in its playlist, and if not, downloads the song via cellular or WiFi. You go out the door, and when you've got your headphones on, you pick up the album where you left off.

Record Chunking

If you need to send content that's larger than the 2^{32}–1 byte limit, you can break a payload into chunks and send it in several records. When you do so, you set the Chunk Flag (one of the TNF flag bits) to 1 for the first chunked record and all following records that are chunked, except the last chunk. You can't chunk content across multiple NDEF messages.

The TNF is set in the first chunked record. Subsequent chunks must use TNF 06 (Unchanged). Middle and terminating chunks must have a *type length* of 0.

The payload length of each record indicates the payload *for that chunk*.

Having a message that exceeds 500MB (that's 2^{32} bits) seems unlikely, so you may not use chunking for that too often. However, chunking can also be used for dynamically generated content, especially where the payload length is not known in advance.

Chunking is used relatively rarely. The libraries used in this book don't implement chunking. Android's parser will read chunked messages and combine them into logical NDEF records.

Additional Info

For more on the structure of NDEF, including a handy set of tests for writing your own NDEF parsing engines, visit the NFC Forum specifications page.

NDEF in Practice

In order to see NDEF in practice, it's useful to look at how existing applications do it. For this section, you'll need to download a few existing apps to your device so you can write some tags with them and compare their work.

One of the most common tasks in many popular tag writer apps is the Foursquare check-in. When you tap a tag formatted with this task, your device will automatically connect to the social media app Foursquare and check you in at a venue. However, each app manages this task slightly differently, and the differences show up in their results.

For this project, you need the following:

- An NFC-enabled Android phone
- Five NFC tags
- The apps that are listed next (you can install these to your device directly from Google Play on your computer or other device)

We also expect that you've gone through Chapter 3 and installed all the software needed to complete the examples in that chapter.

For best results across multiple devices, stick with the NFC Forum tag types; Mifare Classic tags won't work with many newer devices. See "Device-to-Tag Type Matching" on page 19 for more on which tags work with which devices. Avoid Type 2 (Mifare Ultralight) tags as they don't have enough memory for this project.

For this comparison, the following apps were used:

- Trigger by TagStand
- NFC TagWriter by NXP
- NFC Writer, also by TagStand
- TecTiles by Samsung (functional in the US and Canada only)
- App Lancher NFC Tag Writer [sic] by vvakame

You'll also need NXP TagInfo to read the tags back, Foursquare for Android, and a Foursquare account.

Be sure to log in to the Foursquare app before you test any of these. If you run into any problems (such as the Foursquare app flashing on the screen and disappearing), go back to the app and log in again. This is a good example of the sort of problems that can occur entirely outside of your control. After the payload is handed off to Android, an app (such as Foursquare) is responsible for what happens next. If that app doesn't deal with it gracefully, you might be scratching your head.

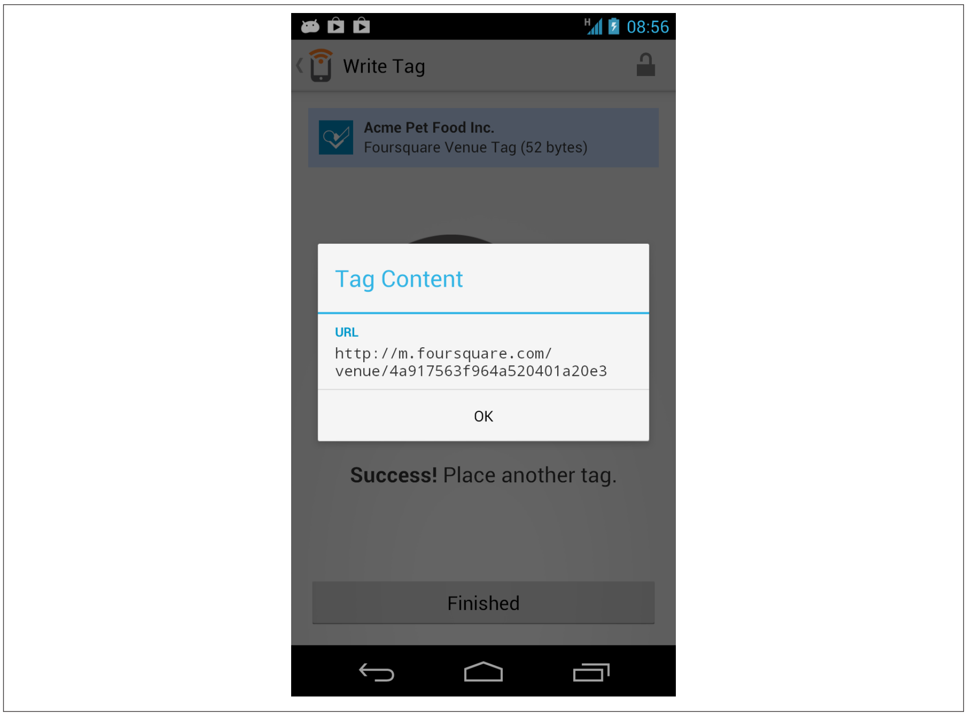
Don't worry if you're not really in the location you're checking into. For one, each app will have to request permission to use Foursquare before actually performing the check-in, and you can always go and delete the check-in on the Foursquare website afterward. There's nothing worse than confusing your friends as to your whereabouts while you're testing NFC!

The process of writing the tags for each of these apps is fairly straightforward, with the exception of TagWriter. Here are the basic steps:

NFC Task Launcher
: Open the app and click the + at the top to make a new task. Choose "NFC" from the task category list. Name your task, then click the + button. From the list, choose "Social Media," then "Foursquare Check-in at a venue." Choose a venue by typing in a name, or click the magnifying glass and the app will look for nearby venues. Click "OK," then click "Add to Task." Click the right arrow to write to tag. Place the phone over the tag, and the app will write to the tag.

TagStand Writer
: Open the app, click "Foursquare Venue," and pick a venue. When the screen changes to the writing screen, click the venue name to see the tag content. Write down the

URL as shown in Figure 4-3 because you'll need it for NXP TagWriter. Place the phone over the tag and the app will write to the tag.



```
http://m.foursquare.com/
venue/4a917563f964a520401a20e3
```

Figure 4-3. When you're on the tagwriting screen of TagStand Writer, you can click on the venue name to see the full URL of the venue

NFC TagWriter

Open the app and choose "Create, Write, and Store." Choose "New," and from the menu, choose "URI." Enter "Foursquare Check-in" in the description field. A description is required for this exercise so that a Smart Poster record is written to the tag. Enter the Foursquare venue as follows: *http://m.foursquare.com/venue/venueid* where *venueid* is a long hexadecimal string. Copy it from the one you wrote down from TagStand Writer previously. Don't forget the *http://*. Click "Next." Place the phone over the tag, and the app will write to the tag.

Samsung TecTiles

Make sure you already have Foursquare installed. Open the TecTiles app and click "New Task." Name it, then click "OK." Click "Add" to create a new task. Click "Application." Choose Foursquare from the list of apps. Click "Write to Tag." Place the phone over the tag, and the app will write to the tag.

AppLauncher NFC
: Open the app, choose Foursquare. Place the phone over the tag, and the app will write to the tag.

Once you've written and labeled five tags, try them out. Close all apps and place the phone over the tag. They should respond as detailed in Table 4-1.

Next, open NXP TagInfo. Place the phone over the tag, and the app will read the tag. Choose NDEF and you can see the NDEF details of any tag (as illustrated in Table 4-1).

Table 4-1. The output of each app

App	Record	TNF	Record type	Payload	Action
NFC Task Launcher	1	MIME	x/nfctl	enZ:Foursquare;c:4a917563f964a520401a20e3	Attempts to launch Task Launcher app; if successful, passes URL to Foursquare app
	2	External	android.com:pkg	com.jwsoft.nfcactionlauncher	
Tagstand Writer:	1	Well-Known	U	*http://m.foursquare.com/venue/4a917563f964a520401a20e3*	Launches Foursquare app, takes you to venue check-in screen
NXP TagWriter	1	Well-Known	Sp		Launches Foursquare app, takes you to venue check-in screen
	1.1		U	*http://m.foursquare.com/venue/4a917563f964a520401a20e3*	
	1.2		T	Foursquare check-in	
Samsung TecTiles	1	Well-Known	U	tectile://www/samsung.com/tectiles	Attempts to launch TecTiles app
	2	Well-Known	T	·enTask····Foursquare·com.joelapenna...[a]	
App Launcher NFC	1	External	android.com:pkg	com.joelapenna.foursquared	Launches Foursquare app only

[a] The Samsung TecTiles message contains unprintable characters. The actual characters are not important and have been replaced with a middle dot. The line was also truncated with an ellipsis. You can see the full byte stream for these tags by writing a tag from TecTiles and reading it with NXP TagInfo.

As you can see, each app does the job differently. There are four basic approaches represented here, however:

- Launch the Foursquare app and let the user do the rest (App Launcher NFC)
- Send a URI and let the operating system do the rest (Tagstand Writer)
- Launch the original app which in turn launches the Foursquare app (NFC Task Launcher, Samsung TecTiles)
- Send a Smart Poster (NXP TagWriter)

The first method just uses one NDEF record, with the TNF set to “External.” The record type is then the Android Application Record for the application you want to launch, and the content is the actual name of the app, like so:

```
TNF: External
Record type: android.com:pkg
com.joelapenna.foursquared
```

Using the first method, you’re telling Android what application to launch only.

The second method also uses just one NDEF record, with the TNF set to “Well-Known” and the record type set to “U” for URI. Again, the content is the actual address, like so:

```
TNF: Well-Known
Record Type: U
http://m.foursquare.com/venue/4a917563f964a520401a20e3
```

Using the second method, you’re telling Android the URI of the thing you want to open and letting the operating system decide what application is best to open it. It’s a bit like letting Windows decide what application opens a file with a particular extension. If Foursquare weren’t on your device, Google Play would open to handle these URLs.

The third method uses an NDEF message composed of two NDEF records. For both NFC Task Launcher and Samsung TecTiles, the original application handles the tag read and then launches Foursquare. NFC Task Launcher uses a MIME-type record that includes the Foursquare venue information and an External AAR record that ensures that the application is installed. TecTiles takes a similar approach with a different implementation. TecTiles uses a URI record with a custom *tectile://* URL to launch the application. It encodes the Foursquare information into a second text record. Unfortunately, TecTiles only launches the application; it does not store the venue information. Both applications use intent filters to launch when their tag is scanned. NFC Task Launcher registers for MIME-type `x/nfctl`. TecTiles registers for their custom *tectile://* URI. You’ll learn more about intent filters in “Android’s Tag Dispatch System” on page 89.

The fourth method uses a Smart Poster record. Smart Posters are a more complex type of NDEF record, where the payload is actually another NDEF message. The message embedded in the Smart Poster payload contains two NDEF records of its own, a URI and a text record. Since Smart Poster records have multiple records, they can deliver additional information about the URI, such as a title, icon, or suggested processing actions.

You can see that some of the applications, like TecTiles and NFC Task Launcher, write Android Application Records to launch their own app, not Foursquare. They then have their app launch Foursquare. This presumably allows them to track when their app is used, even if the end result is to open a different app. It's more complicated, but it arguably allows for gathering metrics about your app's use.

A Tag Writer Application: Foursquare Check-In

In this section, you'll write your own tag writer app in order to get a better understanding of how this works in practice. This app is very simple; it looks for a tag that can be formatted as an NDEF tag, and if it finds one, it writes an NDEF message to the tag.

For this project, you need the following:

- An NFC-enabled Android phone
- Five RFID tags
- Foursquare for Android and a Foursquare account

Start by making a new project as you did in Chapter 3. Use Cordova to create a project, add the Android platform, and install the plug-in:

```
$ cordova create ~/FoursquareCheckin com.example.checkin FoursquareCheckin ❶
$ cd ~/FoursquareCheckin ❷
$ cordova platform add android
$ cordova plugin add https://github.com/chariotsolutions/phonegap-nfc
```

❶ Windows users should type `%userprofile%\FoursquareCheckin` instead of `~/FoursquareCheckin`.

❷ Windows users should type `/d %userprofile%\FoursquareCheckin` instead of `~/FoursquareCheckin`.

Now you're ready to write your app by editing the HTML and JavaScript files. The *index.html* file is in the *www* directory of the app directory you just created and the *index.js* is in *www/js*. Open them both and delete everything to start your own app from scratch. Start with an *index.html* file like so:

```
<!DOCTYPE html>

<html>
   <head>
      <title>Foursquare Check-In Tag Writer</title>
```

```
        <style>body { margin: 20px }</style>
    </head>
    <body>
        <p>Foursquare Check-In Tag Writer</p>

        <div class="app">
            <div id="messageDiv">No tag found</div>
        </div>

        <script type="text/javascript" src="cordova.js"></script>
        <script type="text/javascript" src="js/index.js"></script>
        <script type="text/javascript">
                app.initialize();
          </script>
    </body>
</html>
```

Writing an NDEF Record to a Tag

Next, you're going to write the *index.js* file to format an NDEF record and write an NDEF message to any tag it encounters. For now, you'll keep it simple and hardcode most of the parameters. You've seen many methods for triggering a Foursquare check-in; to start with, just do the simplest thing: launch the Foursquare app using an Android Application Record.

Start with a variable to write when a tag shows up:

```
var app = {
    messageToWrite: [],      // message to write on next NFC event
```

Next comes an `initialize()` function to start things off and a `bindEvents()` function to set up an event listener to detect when the device is ready:

```
    // Application constructor
    initialize: function() {
        this.bindEvents();
        console.log("Starting Foursquare Checkin app");
    },
    /*
        bind any events that are required on startup to listeners:
    */
    bindEvents: function() {
        document.addEventListener('deviceready', this.onDeviceReady, false);
    },
```

After that comes a handler to clear the screen and add an event listener to listen for discovered tags:

```
    /*
        this runs when the device is ready for user interaction:
    */
    onDeviceReady: function() {
      app.clear();

      nfc.addTagDiscoveredListener(
         app.onNfc,             // tag successfully scanned
         function (status) {    // listener successfully initialized
            app.makeMessage();
            app.display("Tap an NFC tag to write data");
         },
         function (error) {     // listener fails to initialize
            app.display("NFC reader failed to initialize "
               + JSON.stringify(error));
         }
      )
   },
```

The NFC event handler, `onNfc()`, will write to the tag, like so:

```
   /*
      called when a NFC tag is read:
   */
   onNfc: function(nfcEvent) {
      app.writeTag(app.messageToWrite);
   },
```

Next come the `display()` and `clear()` functions, just as you wrote them in "PhoneGap Meets NFC: NFC Reader" on page 40:

```
   /*
      appends @message to the message div:
   */
   display: function(message) {
      var label = document.createTextNode(message),
         lineBreak = document.createElement("br");
      messageDiv.appendChild(lineBreak);         // add a line break
      messageDiv.appendChild(label);             // add the text
   },
   /*
      clears the message div:
   */
   clear: function() {
       messageDiv.innerHTML = "";
   },
```

The methods for the `makeMessage()` and `writeTag()` functions that follow make use of functions from two objects defined by the NFC plug-in. These are the NFC object, which gives you access to your device's NFC reader, and the NDEF object, which defines and formats NDEF records and messages.

```
   makeMessage: function() {
      // Put together the pieces for the NDEF message:
      var tnf = ndef.TNF_EXTERNAL_TYPE,           // NDEF Type Name Format
         recordType = "android.com:pkg",          // NDEF Record Type
         payload = "com.joelapenna.foursquared",  // content of the record
```

```
            record,                     // NDEF record object
            message = [];               // NDEF Message to pass to writeTag()

         // create the actual NDEF record:
         record = ndef.record(tnf, recordType, [], payload);
         // put the record in the message array:
         message.push(record);
         app.messageToWrite = message;
      },

      writeTag: function(message) {
         // write the record to the tag:
         nfc.write(
            message,             // write the record itself to the tag
            function () {        // when complete, run this callback function:
               app.display("Wrote data to tag.");   // write to the message div
            },
            // this function runs if the write command fails:
            function (reason) {
               alert("There was a problem " + reason);
            }
         );
      }
   };     // end of app
```

You already know that an NDEF record consists of a TNF, a record type, and a payload. The NDEF object has functions for creating NDEF#Records. An NDEF#Message is simply an array of NDEF#Records. To create a new NDEF record, you need to pass in four parameters:

Type name format
: A 3-bit value. The TNF values are contained as constants in the NDEF object, so you can refer to them using those constants.

Record type
: A string or byte array containing zero to 255 bytes, representing the record type. The NDEF object has built-in constants for many of these as well, though not all, as you can see in this example.

Record ID
: A string or byte array containing zero to 255 bytes. As you read earlier, record IDs are optional, so you can use an empty array, but this value must not be null.

Payload
: A string or array containing zero to (2^{32}–1) bytes. Again, you can use an empty array, but this must not be null.

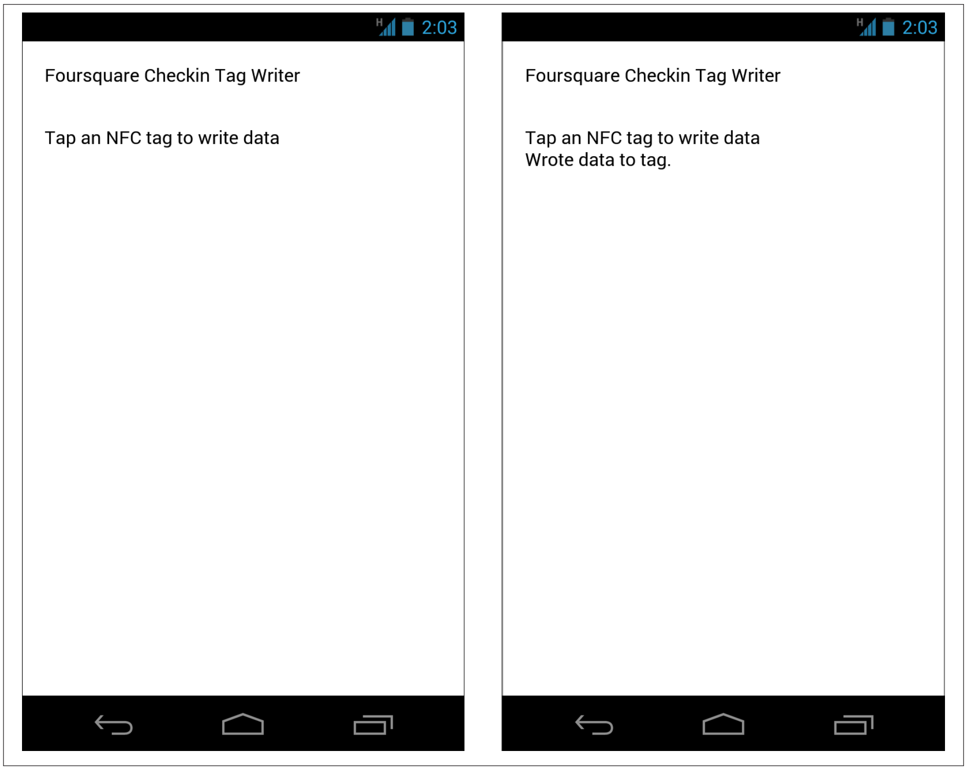
In the `makeMessage()` function, the TNF is `TNF_EXTERNAL_TYPE`, using the constants defined in the NDEF object; the record type is `android.com:pkg`, meaning that your payload will be an Android Application Record; the record ID is not specified, so an empty array is sent; and the payload is the name of the application to launch, `com.joelapenna.foursquared`.

Save both the *index.html* and *index.js* files, then change directories to the root of your new app, and run it:

```
cordova run
```

When you run the app and tap your device to a tag, you should get the tag ID as shown on the left side of Figure 4-4. Click the link and the app will write to the tag and give you the notification seen on the right side of Figure 4-4. Close the app, tap your device to the tag, and it should launch Foursquare.

The full source code can be found on GitHub.



Tap an NFC tag to write data

Tap an NFC tag to write data
Wrote data to tag.

Figure 4-4. The Foursquare check-in app; waiting for a tag (left) and writing to a tag (right)

Writing Different Record Types

Your app can now write a tag to launch another app from that tag, but it doesn't do all the things you saw in the previous examples. It would be great if you could emulate all five of the other apps, for comparison. To do that, make a new project called "FoursquareAdvanced." Install the NFC plug-in, then copy the *index.html* and *index.js* from the previous Foursquare Check-In app example:

```
$ cordova create ~/FoursquareAdvanced com.example.advanced FoursquareAdvanced ❶
$ cd ~/FoursquareAdvanced ❷
$ cordova platform add android
$ cordova plugin add https://github.com/chariotsolutions/phonegap-nfc
$ cp ~/FoursquareCheckin/www/index.html ~/FoursquareAdvanced/www/.
$ cp ~/FoursquareCheckin/www/js/index.js ~/FoursquareAdvanced/www/js/.
```

❶ Windows users should type `%userprofile%\FoursquareAdvanced` instead of `~/FoursquareAdvanced`.

❷ Windows users should type `/d %userprofile%\FoursquareAdvanced` instead of `~/FoursquareAdvanced`.

Start by modifying the *index.html* page by adding a form with an option menu that lets the user choose the app she wants to emulate. Here's the new *index.html* page:

```
<!DOCTYPE html>

<html>
   <head>
    <title>Foursquare Check-In Tag Writer - Advanced</title>
    <style>
        body { margin: 20px }
    </style>
   </head>
   <body>
    <p>Foursquare Check-In Tag Writer - Advanced</p>
    <div class="app">
      <form>
       Write a tag like: <br />
       <select id="appPicker">
         <option value="1">NFC Task Launcher</option>
         <option value="2">TagStand Writer</option>
         <option value="3">NXP TagWriter</option>
         <option value="4">Samsung TecTiles</option>
         <option value="5">App Launcher NFC</option>
       </select>
      </form>
      <div id="messageDiv"></div>
    </div>
    <script type="text/javascript" src="cordova.js"></script>
    <script type="text/javascript" src="js/index.js"></script>
    <script type="text/javascript">
```

```
      app.initialize();
    </script>
  </body>
</html>
```

When the user picks one of the apps in the menu, you'll now have a convenient form element that you can read to determine how to format your tag. Save the *index.html* file, and open your *index.js* file. You'll need to modify `makeMessage()` to handle this. The new function will emulate the apps you saw here by constructing several different types of NDEF messages. First, add a new local variable to read the HTML form element to find out which app the user wants to emulate. The rest of the local variables are the same names, but their values will be generated on the fly this time:

```
makeMessage: function() {
  // get the app type that the user wants to emulate from the HTML form:
  var appType = parseInt(appPicker.value, 10),
      tnf,              // NDEF Type Name Format
      recordType,       // NDEF Record Type
      payload,          // content of the record
      record,           // NDEF record object
      message = [];     // NDEF Message to pass to writeTag()
```

The rest of the `makeMessage()` function will be totally changed. After the local variables, you're going to format a different NDEF message depending on which app you're emulating. Since the HTML form returns an integer from the option menu, you can use that result in a switch-case statement as follows. You can see that each case writes the same records as one of the previous apps. Case 1 creates a MIME media record containing the instructions for the Task Launcher app, then an Android Application Record to tell Android which app to launch:

```
    switch (appType) {
      case 1: // like NFC Task Launcher
        // format the MIME media record:
        recordType = "x/nfctl";
        payload = "enZ:Foursquare;c:4a917563f964a520401a20e3";
        record = ndef.mimeMediaRecord(recordType, payload);
        message.push(record); // push the record onto the message

        // format the Android Application Record:
        tnf = ndef.TNF_EXTERNAL_TYPE;
        recordType = "android.com:pkg";
        payload = "com.jwsoft.nfcactionlauncher";
        record = ndef.record(tnf, recordType, [], payload);
        message.push(record); // push the record onto the message
        break;
```

Case 2 creates a Well-Known type record with the RTD set to URI. Since URIs have a standard format, the NDEF spec includes URI Identifier Codes to be used in place of some of the standard URI headers, to shorten the payload. For example, Table 4-2 shows the first few URI identifier codes.

Table 4-2. Some URI identifier codes

URI Identifier Code (UIC)	What it means
0x00	Nothing, used for custom headers
0x01	*http://www.*
0x02	*https://www.*
0x03	*http://*

The other URI headers you're used to, like *ftp://* and *mailto:* and *file:///*, are all included with their own numbers. For a complete list, see the URI Record Type Definition document, part of the NFC Specification on the NFC Forum site. It's also listed in Appendix A.

In order to add the UIC, you need to convert the URI string to an array of bytes and push the UIC on to the beginning of it. In Case 2, add 0x03, the UIC for *http://*:

```
case 2: // like Tagstand Writer
   // format the URI record as a Well-Known type:
   tnf = ndef.TNF_WELL_KNOWN;
   recordType = ndef.RTD_URI; // add the URI record type
   // convert to an array of bytes:
   payload = nfc.stringToBytes(
      "m.foursquare.com/venue/4a917563f964a520401a20e3");
   // add the URI identifier code for "http://":
   payload.unshift(0x03);
   record = ndef.record(tnf, recordType, [], payload);
   message.push(record); // push the record onto the message
   break;
```

Case 3 is special because it's creating a Smart Poster message. Smart Posters are unique among the Well-Known types in that they're records that contain NDEF messages. As you can see in the next example, you're constructing the payload of the Smart Poster record as an array of records. That array is the message.

In Chapter 5, you'll see how to extract a Smart Poster record by treating its content as a message, and extracting that message recursively:

```
case 3: // like NXP TagWriter
    // The payload of a Smart Poster record is an NDEF message
    // so create an array of two records like so:
    var smartPosterPayload = [
       ndef.uriRecord(
          "http://m.foursquare.com/venue/4a917563f964a520401a20e3"),
       ndef.textRecord("foursquare checkin"),
    ];

    // Create the Smart Poster record from the array:
    record = ndef.smartPoster(smartPosterPayload);
    // push the Smart Poster record onto the message:
    message.push(record);
    break;
```

Case 4 constructs a URI record and a text record that contains some binary data. As you saw in Case 2, you need to convert the URI to a byte array and push the URI Identifier Code on to the front of the array. In this case, since Samsung is using a custom URI header (*tectile://*), use the URI Identifier Code 0x00, which writes the URI exactly as written. TecTiles also writes a second record with TNF 01 (Well-Known type) and type "T."

Samsung is also using a proprietary token in the middle of the payload. You can determine this from reading a tag written with the TecTiles app with the NXP TagReader app. It's duplicated here so that the message is formatted just as the Samsung app wants it to be:

```
case 4: // like TecTiles
    // format the record as a Well-Known type
    tnf = ndef.TNF_WELL_KNOWN;
    recordType = ndef.RTD_URI; // add the URI record type
    var uri = "tectiles://www.samsung.com/tectiles";
    payload = nfc.stringToBytes(uri);
    var id = nfc.stringToBytes("0");
    // URI identifier 0x00 because there's no ID for "tectile://":
    payload.unshift(0x00);
    record = ndef.record(tnf, recordType, id, payload);
    message.push(record); // push the record onto the message

    // text record with binary data
    tnf = ndef.TNF_WELL_KNOWN;
    recordType = ndef.RTD_TEXT;
    payload = [];
    // language code length
    payload.push(2);
    // language code
    payload.push.apply(payload, nfc.stringToBytes("en"));
    // Task Name
    payload.push.apply(payload, nfc.stringToBytes("Task"));
    // 4-byte token proprietary to TecTiles:
    payload.push.apply(payload, [10, 31, 29, 19]);
    // Application Name
    payload.push.apply(payload, nfc.stringToBytes("Foursquare"));
    // NULL terminator
    payload.push(0);
    // Activity to launch
    payload.push.apply(payload, nfc.stringToBytes(
       "com.joelapenna.foursquared.MainActivity"));
    // NULL terminator
    payload.push(0);
    // Application packageName
    payload.push.apply(payload, nfc.stringToBytes(
       "com.joelapenna.foursquared"));
    id = nfc.stringToBytes("1");
    record = ndef.record(tnf, recordType, id, payload);
```

```
         message.push(record); // push the record onto the message
         break;
```

The TecTiles record is too large for certain tags, such as the Mifare Ultralight, which has a limited capacity.

Case 5 constructs an Android Application Record to open Foursquare directly. It's just like all the other Android Application Records you've made already. It's an External TNF, with the record type `android.com:pkg` and the app name as the payload:

```
      case 5: // like App Launcher NFC
         // format the Android Application Record:
         tnf = ndef.TNF_EXTERNAL_TYPE;
         recordType = "android.com:pkg";
         payload = "com.joelapenna.foursquared";
         record = ndef.record(tnf, recordType, [], payload);
         message.push(record); // push the record onto the message
         break;
   } // end of switch-case statement
```

Finally, finish off the `makeMessage()` function by setting the message to write and notifying the user:

```
   app.messageToWrite = message;
   app.display("Tap an NFC tag to write data");
}, // end of makeMessage()
```

With these changes you can save the *index.js* file and run the app (make sure you still have the `writeTag()` function in your app from the previous example):

```
$ cordova run
```

When you do, you'll get a drop-down menu that lets you emulate any of the apps you saw in Table 4-1. To use it, pick an app to emulate from the menu, then tap the phone to a tag. Figure 4-5 shows the app.

The full source code can be found on GitHub.

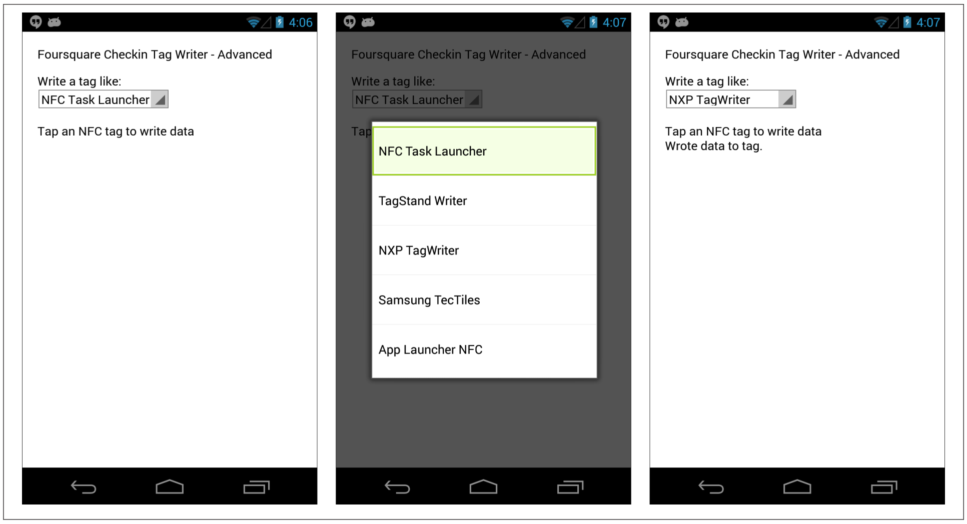


Figure 4-5. The Foursquare Check-In Advanced app; initial screen (left), showing options menu (center), and after writing to tag (right)

PhoneGap-NFC NDEF Helper Functions, Summarized

As you can see, this app now covers multiple functions of the PhoneGap-NFC library by emulating various approaches to the same task. There are a few helper functions in the NDEF object it contains that can be used to write NDEF records. The main one is `ndef.record()`, which requires you to give it the TNF, record type, ID, and payload:

```
// Caller specifies the TNF and record type
record = ndef.record(tnf, recordType, id, payload);
// example:
record = ndef.record(ndef.TNF_EXTERNAL_TYPE, "android.com:pkg",
   [],"com.joelapenna.foursquared");
```

There's also a helper for MIME media records that takes just a MIME type and the payload:

```
// TNF: MIME media (02)
record = ndef.mimeMediaRecord(mimeType, payload);
// example:
record = ndef.mimeMediaRecord("text/json", '{"answer": 42}');
```

The URI helper builds a record using TNF 01, Well-Known, and record type "U" to send a URI as you saw in Case 3 of "Writing Different Record Types" on page 66. The helper will shorten your URI using a URI Identifier Code. In this example, *http://* will be written to the tag as 0x03 followed by *m.foursquare.com*:

```
// TNF: Well-Known Type(01), RTD: URI ("U")
record = ndef.uriRecord(uri);
```

```
// example:
record = ndef.uriRecord("http://m.foursquare.com/");
```

The text record helper makes text records using a Well-Known TNF and a text record type definition. If no language is specified, it defaults to English:

```
// TNF: Well-Known Type(01), RTD: Text ("T")
record = ndef.textRecord(text, language);
// example:
// defaults to English, since no language specified:
record = ndef.textRecord("How are you doing?");
```

The Smart Poster helper constructs a Smart Poster from other records:

```
//TNF: Well-Known Type (01), RTD: Smart Poster ("Sp")
record = ndef.smartPoster(ndefMessage);
// example:
record = ndef.smartPoster (
   // URI record:
   ndef.uriRecord("http://m.foursquare.com/venue/4a917563f964a520401a20e3"),
   // text record:
   ndef.textRecord("foursquare checkin")
);
```

There's also a helper you didn't see in this chapter, the empty record helper. This just makes an empty record for you to fill out:

```
// TNF: Empty (00)
record = ndef.emptyRecord();
// example:
record = ndef.emptyRecord();  // it's empty!
```

With these functions, you can construct all of the types of NDEF messages you'll see in the rest of this book.

Conclusion

We'll go further into NDEF records and messages in the following chapters. The key concepts to take away from this introduction, though, are as follows:

NFC-formatted *tags* contain NDEF *messages*. NDEF messages are composed of one or more NDEF *records*. You'll be tempted to talk about tags and messages and records interchangeably, but don't do it. Later in the book, you'll pass messages from device to device with no tags.

All NDEF records have a type. In the specs, its sometimes called Payload type, sometimes called Record type, other times just type. The type name format (TNF) categorizes the types broadly into a few areas and tells you how to interpret the type:

For TNF 01 (Well-Known)
: The NFC Forum Record Type Definition (RTD) specification tells you what types to use.

For TNF 02 (MIME)
: The MIME RFC (RFC 2046) tells you what types are valid.

For TNF 03 (URI)
: The URI RFC (RFC 3986) tells you how to make valid types.

For TNF 04 (External)
: The NFC Forum RTD specification tells you how to make your own types.

There are eight formal type name formats of NDEF messages, but most of the work gets done with just three: Well-Known, MIME media, and External. The Well-Known TNF includes a number of useful definitions for record types, including text messages, URIs, Smart Posters, and the various carriers needed for peer-to-peer handovers. You'll learn more about the peer-to-peer messages in Chapter 8. You'll see text messages and URIs used throughout this book. The MIME media TNF covers all Internet media types, and you can use it to make custom types as well, as you'll see in Chapter 6. The External TNF is used for records like the Android Application Record.

You won't see a lot of Smart Poster types in this book. As you saw in the Smart Poster case in "Writing Different Record Types" on page 66, the payload of a Smart Poster record is itself an NDEF message. So in order to read a Smart Poster, you first have to parse the message that encloses it, then you have to parse the message that it encloses. We feel this is redundant for most applications, and that it's better practice to simply use multiple records in an NDEF message.

There are multiple ways to achieve the same ends with NDEF messages. How that message is received depends on the operating system of the device that's receiving it. In the next chapter, you'll look at how Android offers you a few different ways to filter NDEF types to your application.

CHAPTER 5

Listening for NDEF Messages

Every well-designed application relies on good listening for relevant events, whether they're user input events, network events, or some other form. NFC applications are no different. NFC applications are always listening for NDEF messages, parsing and filtering the message type and contents, and taking action accordingly. In Chapter 4, you got an introduction on how to write NDEF messages using the PhoneGap-NFC plug-in, but we didn't dive deeply into the listening part.

There are two main ways to listen for NDEF messages: you can program your application to listen for them when it's the app in the foreground, or you can let the operating system of your device do the listening, and call your application when it sees a message that's relevant to your app. The PhoneGap-NFC plug-in offers several NFC events that you can program your app to listen for, and Android offers a *tag dispatch system* that allows you to tell the operating system which tags you're most interested in. NFC apps can take advantage of the tag dispatch system by writing *intent filters* in the Android Manifest that tell the operating system which tags to route to the app. They can also explicitly listen for some or all tags, overriding the intent filters of other apps using the *foreground dispatch system*. In this chapter, you'll learn a bit more about how to listen for and filter NDEF messages using PhoneGap-NFC's event listeners and Android's tag dispatch system.

PhoneGap-NFC's Event Listeners

Up until now, you haven't specified how Android should deal with tags, and you've used only one event listener, the `tagDiscoveredListener`. This event listener uses the foreground dispatch system to tell Android that you want your app to receive all NFC-related messages, and not to route them to other apps. It works well when you want everything to go through your app, and you've probably figured out already that some of the other apps you've seen, like TagWriter and TagInfo, also use the foreground dispatch system in this manner. When your app is in the foreground, it gets priority notification of all events, NFC and otherwise. That means it's up to you to decide what you want to listen for and what to ignore.

The PhoneGap-NFC plug-in offers you four different NFC-related event listeners:

Tag discovered listener
: Listens for all tags that are compatible with the reader hardware. This is the most general listener.

NDEF-formatable listener
: Listens for all compatible tags that can be formatted to receive NDEF messages.

NDEF listener
: Listens for any tags containing NDEF messages. If a legitimate NDEF message is received, this listener will generate an event.

MIME-type listener
: Listens only for those NDEF messages containing a MIME-type. This is the most specific of all the event listeners, and often the most useful. You can use the MIME-type listener to filter for messages containing a specific MIME-type. It will then ignore messages containing other MIME-types. You can also use it to filter for messages containing no type at all. You can't, however, register two MIME-type listeners for different types.

MIME Media-Types in NDEF

The word *type* has a lot of meanings when talking about NFC. If you're still confused, review "Type Name Format" on page 51 and "Payload Type" on page 51, which cover it in detail. Much of this chapter focuses on one particular type, the MIME media-type. *MIME* (originally *Multipurpose Internet Mail Extensions*) is an Internet standard originally designed to extend email to support various nontext media types. For a full list of MIME-types, see the Internet Assigned Numbers Authority (IANA) document on MIME types. The NFC Forum specification writers realized that they didn't need to reinvent a media type protocol, because MIME already did a pretty good job.

For NDEF records with TNF MIME media-type, you generally use one of the preexisting MIME-types like `text/plain` as the record type to describe the contents of the payload. The NDEF specification says "The use of nonregistered media types is discouraged." We do it anyway. Why? MIME-types are an easy, convenient way to differentiate your NFC tags from other tags. When using the Hue lights in Chapter 6, for example, we define a MIME-type of `text/hue`. You also saw it in "NDEF in Practice" on page 56 when NFC Task Launcher used a type called `x/nfctl`. That's not a type you'll find in the general MIME specification.

Android provides intent filters (see "Android's Tag Dispatch System" on page 89 further on in this chapter) that allow an application to register to listen for NFC tags it cares about. When the operating system encounters an NFC tag that your app is registered to listen for, it will launch your application and include the data from the NFC tag. NFC Task Launcher is likely registered to listen for the MIME-type `x/nfctl`. You'll do something similar when you get to Chapter 6.

You can register multiple listeners in the same app, and Android will generate only the most specific event if a given tag satisfies more than one listener. For example, a tag with a plain-text message on it could potentially trigger all of these listeners. It's a readable tag, it's NDEF-formatable, it's NDEF-formatted, and it has a legitimate MIME type (`text/plain`). However, since the MIME type is the most specific listener, that's the event that would be generated.

You can use multiple cascading listeners. For example, if you knew that most of your tags would be plain-text tags, you could have a specific MIME-type listener that captures them, then use an NDEF listener to capture all other NFC types, and an NDEF-formatable listener to search for empty tags and prompt the user to put something on them.

The next application shows this in action. In this app, you'll implement all four listeners, and use different tags to trigger each one.

An NDEF Reader Application

For this project, you need the following:

- An NFC-enabled Android phone
- At least four NFC tags (for best results across multiple devices, stick with the NFC Forum types; see "Device-to-Tag Type Matching" on page 19 for more on which tags work with which devices).

We also expect that you've gone through Chapter 3 and installed all the software needed to complete the examples in that chapter.

Start by making a new project using the `cordova create` command:

```
$ cordova create ~/NdefReader com.example.ndefreader NdefReader ❶
$ cd  ~/NdefReader ❷
$ cordova platform add android
$ cordova plugin add https://github.com/chariotsolutions/phonegap-nfc
```

❶ Windows users should type `%userprofile%\NdefReader` instead of `~/NdefReader`.

❷ Windows users should type `/d %userprofile%\NdefReader` instead of `~/NdefReader`.

You can copy the *index.html* and *index.js* from the NFC Reader app as a base for this project:

```
$ cp ~/NfcReader/www/index.html ~/NdefReader/www/.
$ cp ~/NfcReader/www/js/index.js ~/NdefReader/www/js/.
```

The *index.html* for this project should look like this:

```
<!DOCTYPE html>

<html>
   <head>
      <title>NDEF Reader</title>
   </head>
   <body>
      <div class="app">
         <p>NDEF Reader</p>
         <div id="messageDiv"></div>
      </div>
      <script type="text/javascript" src="cordova.js"></script>
      <script type="text/javascript" src="js/index.js"></script>
      <script type="text/javascript">
       app.initialize();
     </script>
    </body>
</html>
```

Listening for Multiple Events

The *index.js* file starts with the same base as the NFC Reader app. The `initialize()` and `bindEvents()` functions will remain the same as well as the `display()` and `clear()` functions. The `onDeviceReady()` function will start the same, but you'll add three new listeners after the `addTagDiscoveredListener`. The new function will look like this:

```
/*
   this runs when the device is ready for user interaction:
*/
   onDeviceReady: function() {

      nfc.addTagDiscoveredListener(
         app.onNonNdef,           // tag successfully scanned
         function (status) {      // listener successfully initialized
            app.display("Listening for NFC tags.");
         },
         function (error) {       // listener fails to initialize
            app.display("NFC reader failed to initialize "
               + JSON.stringify(error));
         }
      );

      nfc.addNdefFormatableListener(
         app.onNonNdef,           // tag successfully scanned
```

```
            function (status) {      // listener successfully initialized
               app.display("Listening for NDEF Formatable tags.");
            },
            function (error) {       // listener fails to initialize
               app.display("NFC reader failed to initialize "
                  + JSON.stringify(error));
            }
         );

         nfc.addNdefListener(
            app.onNfc,               // tag successfully scanned
            function (status) {      // listener successfully initialized
               app.display("Listening for NDEF messages.");
            },
            function (error) {       // listener fails to initialize
               app.display("NFC reader failed to initialize "
                  + JSON.stringify(error));
            }
         );

         nfc.addMimeTypeListener(
            "text/plain",
            app.onNfc,               // tag successfully scanned
            function (status) {      // listener successfully initialized
               app.display("Listening for plain text MIME Types.");
            },
            function (error) {       // listener fails to initialize
               app.display("NFC reader failed to initialize "
                  + JSON.stringify(error));
            }
         );

         app.display("Tap a tag to read data.");
      },
```

As you can see, these listeners are all mostly the same in their structure. The only change is the message they display when they're successfully registered, and the callback function they invoke. Those that are likely to return an NDEF message call `onNfc()`, and those that are not call `onNonNdef()`.

When the tag you read is not compatible with NDEF in any way, the listener doesn't return a tag. For example, you might read 13.56MHz RFID tags that are not even Mifare tags (Philips ICODE or Texas Instruments Tag-it); your reader can read its UID but nothing else. If you're using a device like the Nexus 4, which can't read Mifare Classic tags, you'll only be able to read the UID but no tag data. You need to handle these tags differently. The `onNonNdef()` handler takes care of this case.

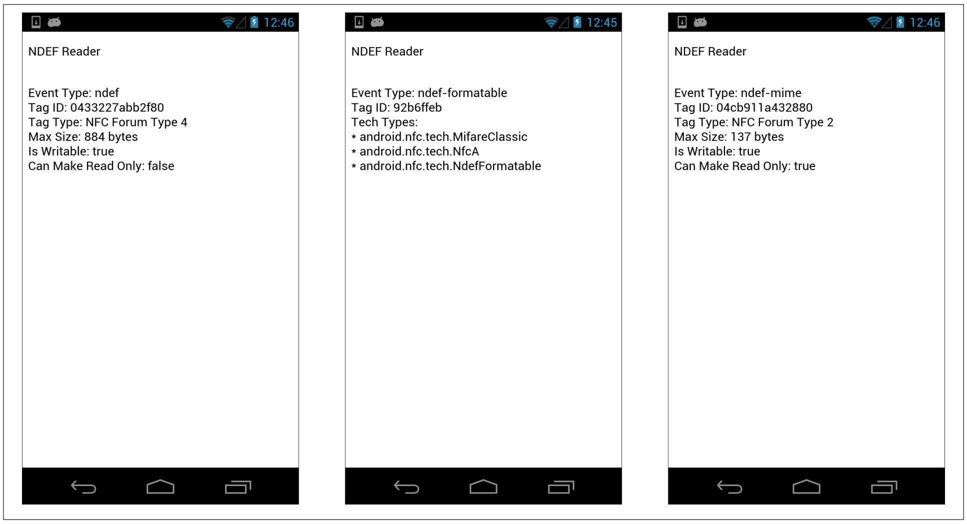
The new `onNfc()` function will call `clear()` to clear the message div, then display the type of event that called it, and the details about the tag. The `onNonNdef()` function will read the tag UID and the tech types from the tag. Then add a `showTag()` function to show the tag details from `onNfc()`:

```
    /*
       Process NDEF tag data from the nfcEvent
    */
    onNfc: function(nfcEvent) {
       app.clear();              // clear the message div
       // display the event type:
       app.display(" Event Type: " + nfcEvent.type);
       app.showTag(nfcEvent.tag);   // display the tag details
    },

    /*
       Process non-NDEF tag data from the nfcEvent
       This includes
        * Non NDEF NFC Tags
        * NDEF-Formatable Tags
        * Mifare Classic Tags on Nexus 4, Samsung S4
        (because Broadcom doesn't support Mifare Classic)
    */
    onNonNdef: function(nfcEvent) {
       app.clear();              // clear the message div
       // display the event type:
       app.display("Event Type: " + nfcEvent.type);
       var tag = nfcEvent.tag;
       app.display("Tag ID: " + nfc.bytesToHexString(tag.id));
       app.display("Tech Types: ");
       for (var i = 0; i < tag.techTypes.length; i++) {
          app.display("  * " + tag.techTypes[i]);
       }
    },

 /*
    writes @tag to the message div:
 */

    showTag: function(tag) {
       // display the tag properties:
       app.display("Tag ID: " + nfc.bytesToHexString(tag.id));
       app.display("Tag Type: " +  tag.type);
       app.display("Max Size: " +  tag.maxSize + " bytes");
       app.display("Is Writable: " +  tag.isWritable);
       app.display("Can Make Read Only: " +  tag.canMakeReadOnly);
     },
```

That gives you enough to run the app. Save all your files and run the app as usual:

```
$ cordova run
```

Try this app with the Foursquare check-in tags you made for Chapter 4. Then try making a new tag, with a plain-text message. You can do this using NXP TagWriter. Tap "Create, write, and store," then "New," then "Plain Text," then enter a text message. Finally, tap "Next," then tap a new tag to the back of your device to write to it. Then re-open the NDEF Reader app you just wrote, and read this tag. It should read as an ndef-mime event, unlike the others. Figure 5-1 shows a few different results you should get from this app so far.



```
NDEF Reader

Event Type: ndef
Tag ID: 0433227abb2f80
Tag Type: NFC Forum Type 4
Max Size: 884 bytes
Is Writable: true
Can Make Read Only: false
```

```
NDEF Reader

Event Type: ndef-formatable
Tag ID: 92b6ffeb
Tech Types:
* android.nfc.tech.MifareClassic
* android.nfc.tech.NfcA
* android.nfc.tech.NdefFormatable
```

```
NDEF Reader

Event Type: ndef-mime
Tag ID: 04cb911a432880
Tag Type: NFC Forum Type 2
Max Size: 137 bytes
Is Writable: true
Can Make Read Only: true
```

Figure 5-1. Results from the NDEF Reader app: a generic NDEF-formatted tag; a blank, but NDEF-formatable tag; and a text tag with MIME-type "text/plain"

You probably didn't get any new events of type "tag." That's because it's overridden by every other event listener. It's rare you'll run across a tag that's compatible with the NFC reader, but not at least NDEF-formatable. Certain non-Mifare RFID tags will show up this way (including Texas Instrument Tag-it HF tags, and one of Don's credit cards), but not much else. If you happen to write PhoneGap-NFC apps for BlackBerry 7, however, you'll see that all the eligible listeners will fire when a given tag comes into range, so you'll have to adapt your code for cross-platform use if you want BlackBerry users to get the same results as Android users. For more on cross-platform differences, see the PhoneGap-NFC plug-in README file.

Reading the NDEF Messages

Now that you've got an app that can read any type of compatible tag, you might as well extend it to give you the details of the NDEF messages on each tag it reads. Keep in mind NDEF messages are made of NDEF records, and when the record is a Smart Poster, the contents of the record is itself an NDEF message. In order to handle this, you'll need a `showMessage()` function that's called by `showTag()`, and a `showRecord()` function that's called by showMessage(). If the record is a Smart Poster, you'll need to call showMessage() again. This is where recursion is your friend, as you'll see. Figure 5-2 shows you the flow of the whole program.

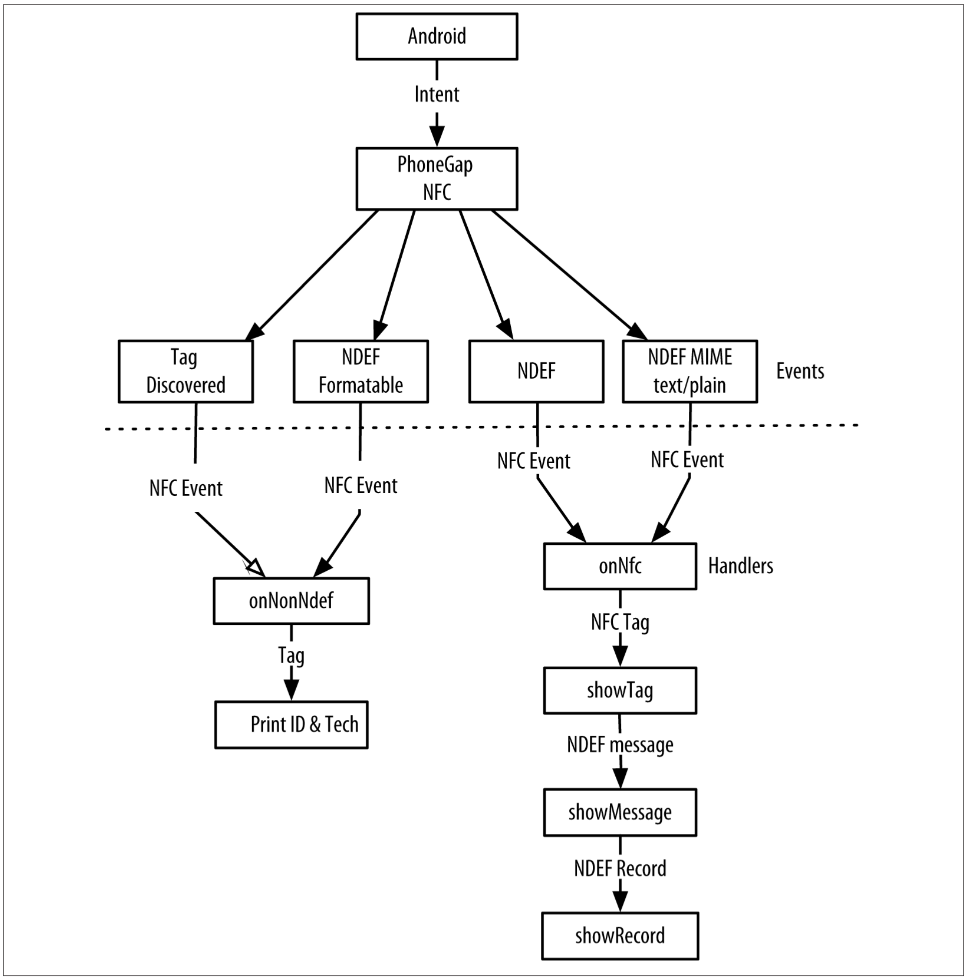


Figure 5-2. The program flow for the NDEF Reader app

Start by adding an if statement to the showTag() function that sends the NDEF message on the tag to the showMessage() function:

```
showTag: function(tag) {
    // display the tag properties:
    app.display("Tag ID: " + nfc.bytesToHexString(tag.id));
    app.display("Tag Type: " +  tag.type);
    app.display("Max Size: " +  tag.maxSize + " bytes");
```

```
        app.display("Is Writable: " +  tag.isWritable);
        app.display("Can Make Read Only: " +  tag.canMakeReadOnly);

        // if there is an NDEF message on the tag, display it:
        var thisMessage = tag.ndefMessage;
        if (thisMessage !== null) {
           // get and display the NDEF record count:
           app.display("Tag has NDEF message with " + thisMessage.length
              + " record" + (thisMessage.length === 1 ? ".":"s."));

           app.display("Message Contents: ");
           app.showMessage(thisMessage);
        }
     },
```

Next, add a function `showMessage()` to show the message. The message is just an array of records, so iterate over it to show each record using a second function, `showRecord()`:

```
  /*
     iterates over the records in an NDEF message to display them:
  */
     showMessage: function(message) {
        for (var thisRecord in message) {
           // get the next record in the message array:
           var record = message[thisRecord];
           app.showRecord(record);          // show it
        }
     },
  /*
     writes @record to the message div:
  */
     showRecord: function(record) {
        // display the TNF, Type, and ID:
        app.display(" ");
        app.display("TNF: " + record.tnf);
        app.display("Type: " +  nfc.bytesToString(record.type));
        app.display("ID: " + nfc.bytesToString(record.id));

        // if the payload is a Smart Poster, it's an NDEF message.
        // read it and display it (recursion is your friend here):
        if (nfc.bytesToString(record.type) === "Sp") {
           var ndefMessage = ndef.decodeMessage(record.payload);
           app.showMessage(ndefMessage);

        // if the payload's not a Smart Poster, display it:
        } else {
           app.display("Payload: " + nfc.bytesToString(record.payload));
        }
     }
  };     // end of app
```