Developing NFC Applications on the Android Platform

The Definitive Resource

Part 1

By Kyle Lampert

Introduction

- This guide will use examples from Mac OS X, but the steps are easily adaptable for modern versions of Linux and Windows.
- In this quick-start guide, you will learn how to set up your environment for Android development, set up a simple application, implement Android Beam (Peer-to-peer NFC transfer), read and write to NFC Tags, interface with a social network, and persist application data.
- Many of the tutorials contained in this document will be geared towards developing for Android 4.0 and higher; if you are targeting devices with prior versions of Android, you will notice that some Android APIs which are referenced in this guide may not be available during your development.
- These tutorials are geared towards developers with no prior experience in Android development. Experienced developers will be able to skip the 'Getting Started' section and jump right into the NFC-specific tutorials

Table of Contents

Getting Started

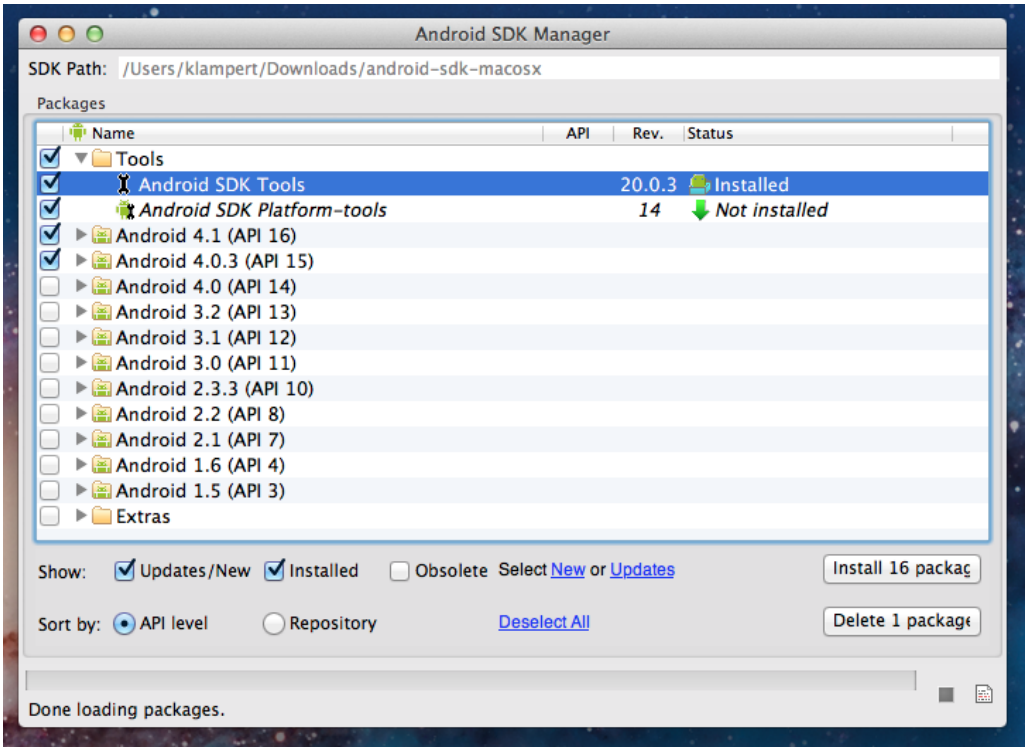
Download and Install

Java

1. Download and install the Java Development Kit and Java Runtime Environment from Oracle if your environment does not currently have them installed.
    a. http://www.oracle.com/technetwork/java/javase/downloads/index.html
2. Ensure that the *bin* directory in the Java installation exists in your PATH environmental variable. Running java -version should output something similar to *java version "1.6.0_31"*

Android SDK

1. Visit the Android developer website to download the Android SDK
    a. http://developer.android.com/sdk/index.html
2. Ensure that your operating system meets the minimum system requirements
3. Install the SDK:
    a. Windows users will install from an executable package
    b. Mac and Linux users should unzip the zip into a location that is easily accessible from the command line
4. Although the SDK has been downloaded, you must use the SDK manager to download AVDs (Android Virtual Device images) and additional platform tools
5. Access the SDK manager:
    a. On Windows, open via the Start Menu and clicking “Run as Administrator” (Note: You should always open the SDK and AVD managers by choosing this option).
    b. On Mac or Linux, run *{path to sdk}/tools/android* from the command line
6. In the SDK Manager, select the checkboxes next to the following items (also pictured below)
    a. Android SDK Tools
    b. Android SDK Platform tools
    c. Andorid 4.1, API 16 (optional - allows for an emulator to be run on your computer)
    d. Andorid 4.0.3, API 15 (optional)



7. Click “Install Packages” and wait for the installation to complete
8. If you wish to create an AVD (Android Virtual Device) to quickly test code changes, open the AVD manager.
    a. From the “Tools” menu of the SDK Manager, click “Manage AVDs”
9. Mac and Linux users may find it useful to add the Android SDK’s *tools* directory to your PATH system variable for easy access to the SDK’s tools

Eclipse

Eclipse is the recommended IDE for Android development.

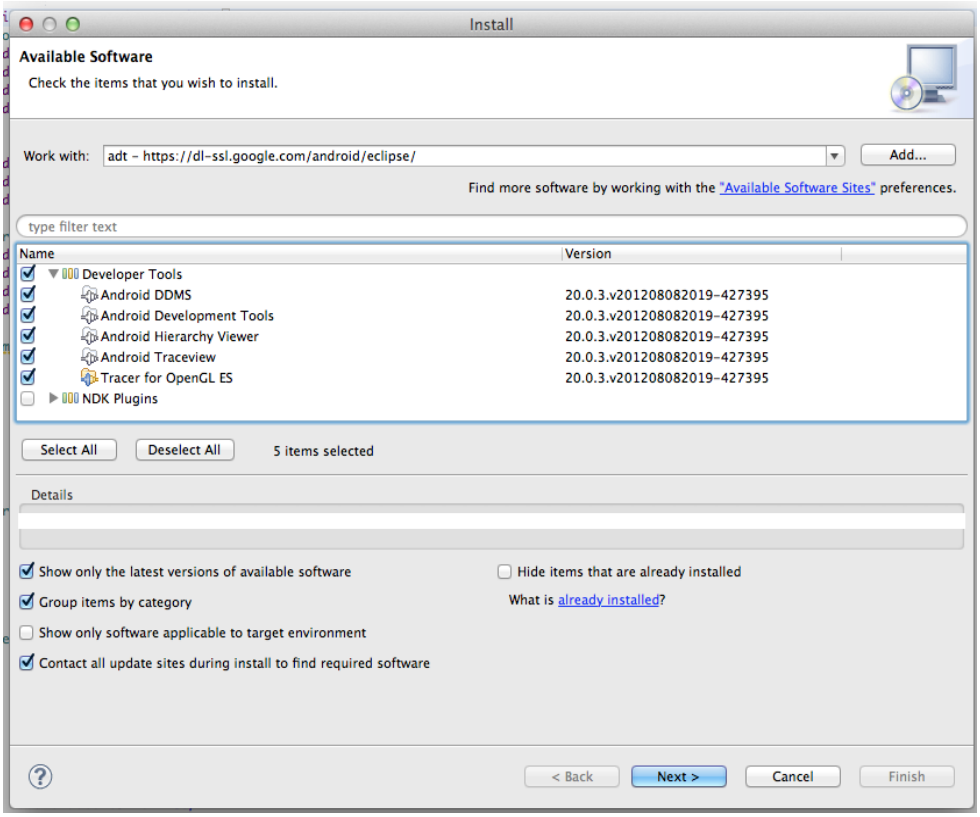
1. Download the latest version of Eclipse Classic. *Note: As of October 2012, Eclipse 3.6.2 or greater is required to work with the Android Development Tools plugin*
    a. http://www.eclipse.org/downloads/
2. Upon first opening Eclipse, you will be prompted to select a workspace. Choose a convenient location - this is the directory in which your development projects will be stored within.
3. From the Eclipse ‘Welcome’ splash screen, click the “Workbench” button.
4. Install the ADT plugin by following the directions below.

ADT (Android Development Tools) Plugin

Developing Android applications in Eclipse is greatly simplified by installing the ADT plugin. This powerful plugin bridges your IDE and Android SDK, allowing for quick project setup, fast UI

development, and access to the Android Debug Bridge (ADB) for running and debugging your application.

1. In Eclipse click “Help” in the main menu, and choose “Install New Software”
2. In the top right corner of the dialog that appears, click “Add” and enter the following information
    a. Name: ADT Plugin
    b. Location: https://dl-ssl.google.com/android/eclipse/
3. If you encounter an error at this point, try using ‘http’ rather than ‘https’ in the Location field
4. On the ‘Available Software’ screen, select all options under ‘Developer Tools’

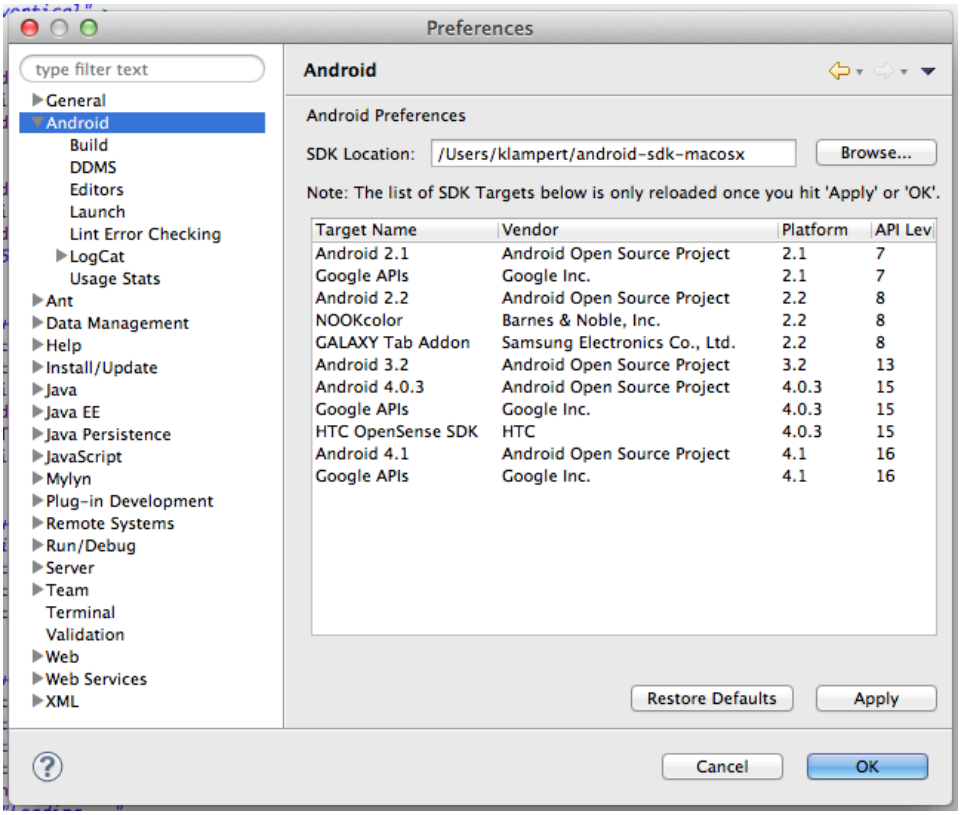


5. Click through and accept the license agreements. It is safe to ignore any warnings about the authenticity of the software.
6. Restart Eclipse to complete the installation.

Configure ADT

1. Once you’ve restarted Eclipse, set the path of the SDK in your Eclipse preferences
    a. On Windows, click Window > Preferences
    b. On Mac OS X, select Eclipse > Preferences

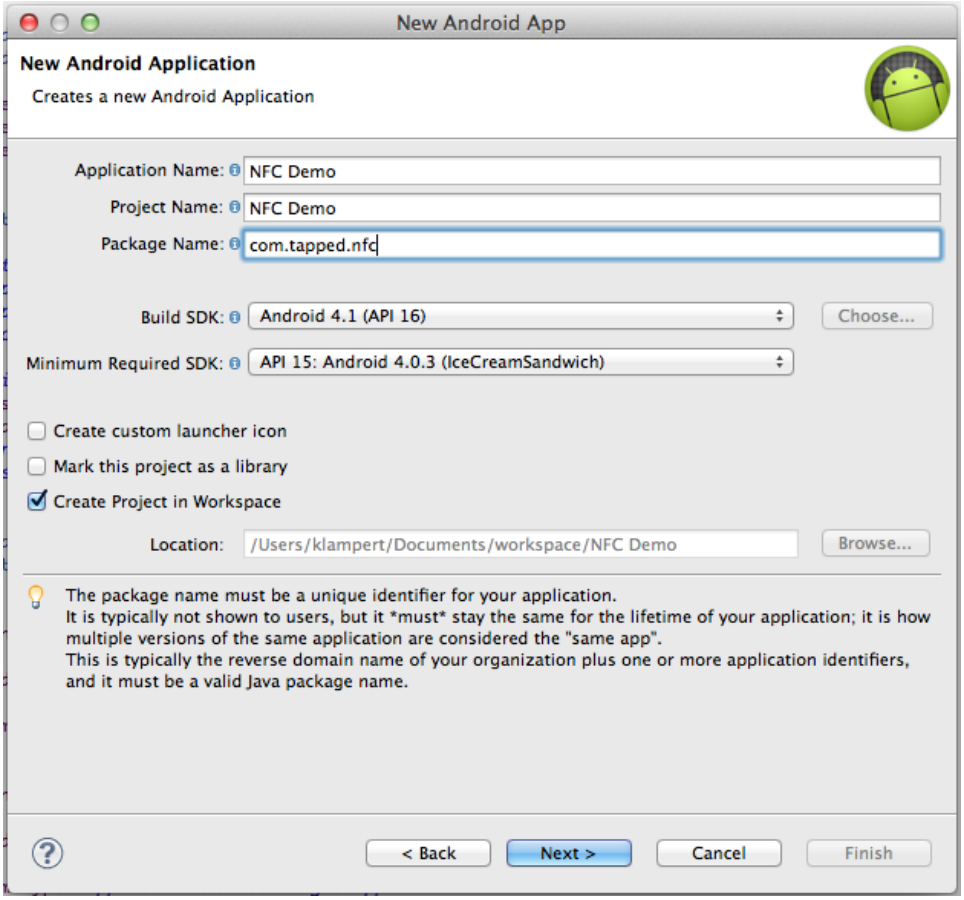
2. Find ‘Android’ on the left panel. You should see a field labeled ‘SDK Location’ in the main preferences window.
3. Click ‘Browse’ next to the SDK Location field and navigate to the root directory of your SDK installation



Creating a Project

New Project

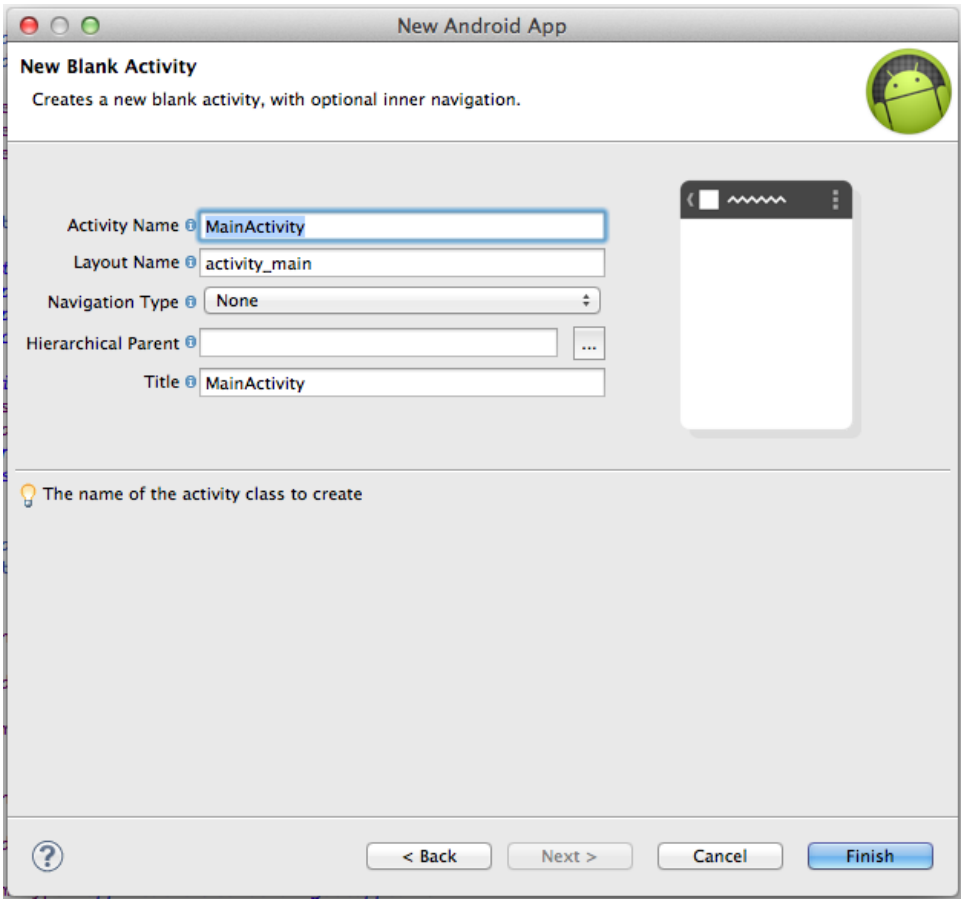
1. To create a new Android Project in Eclipse, select *File* > *New* > *Android Application Project.* **Note**: You may have to choose “Other” from the File > New menu to see Android options
2. Fill out the required fields on the following screen, including the Application and Project names, the package name, and the SDK Versions (pictured below)



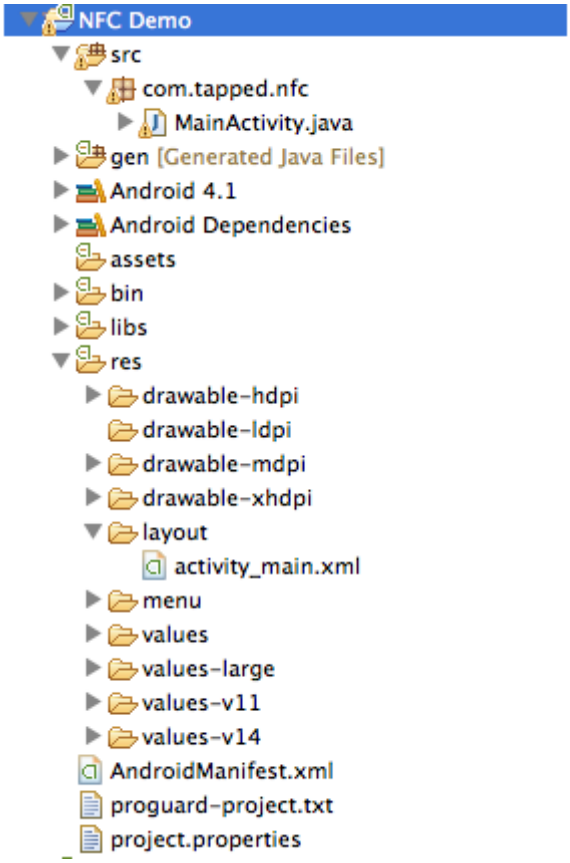
3. Click ‘Next’ to continue to the ‘Create Activity’ screen. Select the first option, labeled “BlankActivity.”

(Aside) What is an Activity? Google explains that an Activity is *“a single screen with a user interface... an email application might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. Although the activities work together to form a cohesive user experience in the email application, each one is independent of the others.”* Additionally, it is important to understand that each Activity has a well-defined lifecycle that corresponds to events in the user experience. To read about more fundamental application concepts, read ‘Application Fundamentals’ on Google’s Android Developer portal http://developer.android.com/guide/components/fundamentals.html

4. Select the default options when creating your first activity. Each of these options will be revisited and explained in greater detail at a later point in this document.



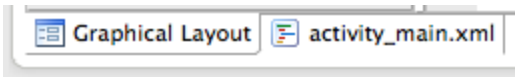
5. Once the new project has been added to your workspace in Eclipse, you should end up with a project hierarchy that looks similar to the outline pictured below. There are three key components within the project structure that you will primarily be working with:
    a. Java files located in *src* > *{package name}*
    b. XML Layout resource files located in *res* > *layout*
    c. The Android Manifest XML file located in the root directory



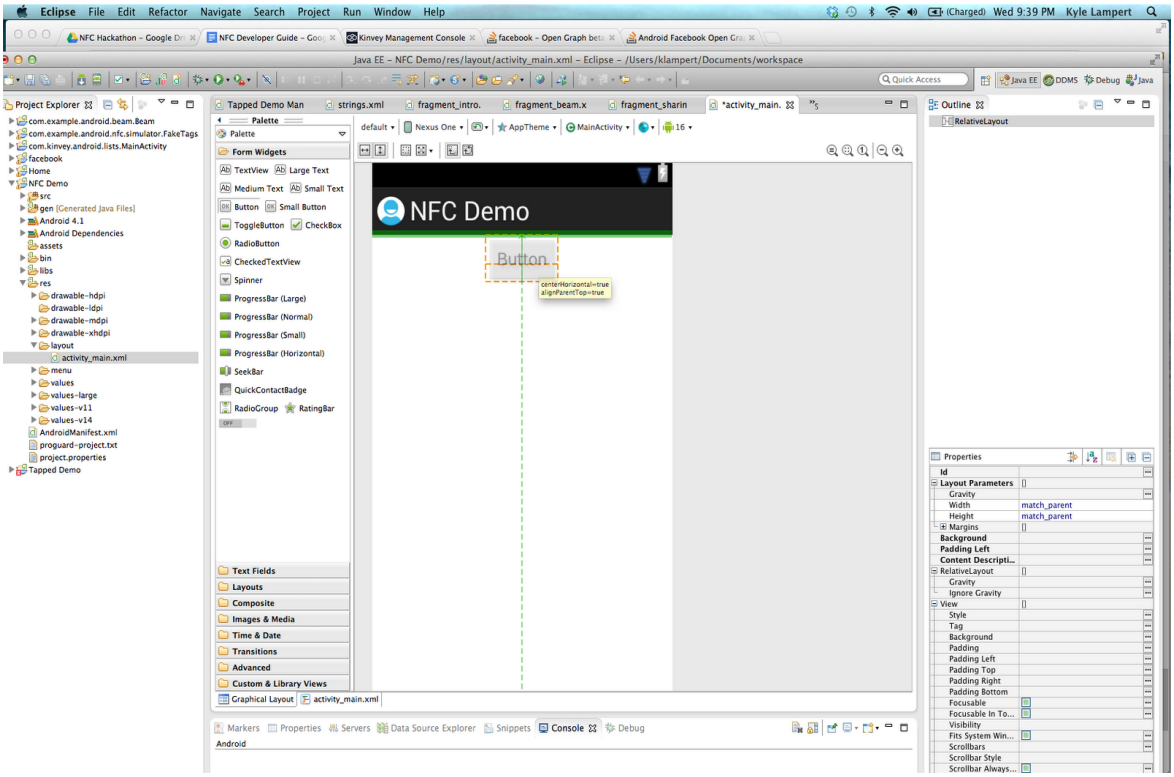
(Aside) What are Layout resource files? Google's Android documentation describes a layout resource file as *"the architecture for the user interface in an Activity. It defines the layout structure and holds all the elements that appear to the user."* These markup files contain a hierarchy of view elements with a similar nesting to that of other markup languages such as HTML. Each activity will specify a layout resource file and programmatically define the detailed behavior of the UI elements defined in the layout.

Editing a Layout File

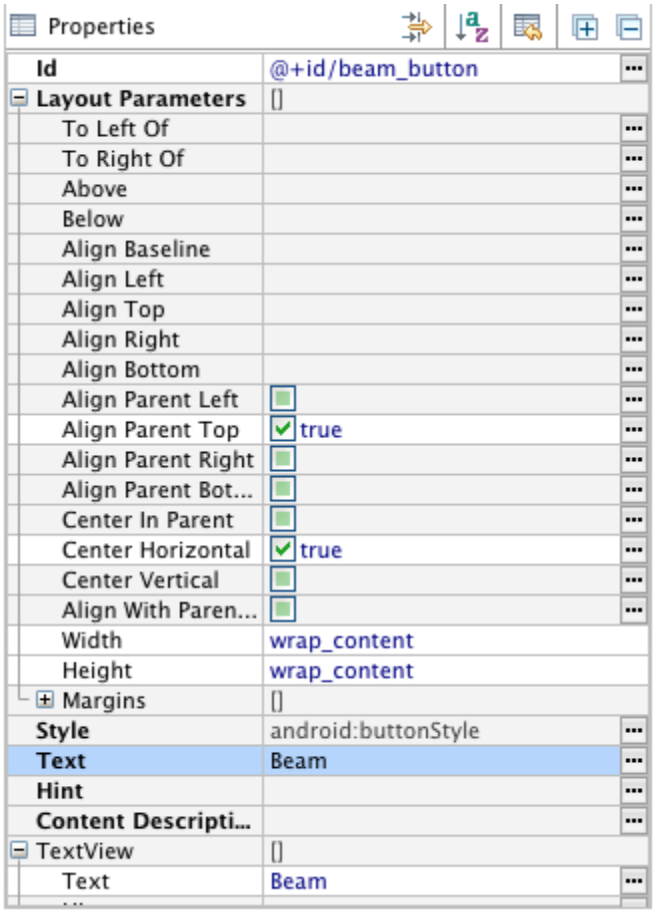
1. Double click on the "activity_main.xml" layout file in the *res* > *layout* directory if it has not already been opened by Eclipse. Underneath the editor you will notice there are two tabs for modifying this file:
    d. Graphical Layout: A "what-you-see-is-what-you-get" graphical editor that allows you to drag-and-drop UI elements onto your screen and modify their properties in a straight-forward manner. This is often the simplest way to get started with designing layouts for an Android application.
    e. XML Editor: In some cases it is more desirable to edit the XML directly; this tab will allow you write your own code or to fine-tune the results of the graphical editor.



2. In the graphical editor, you will likely have a pre-populated text field in the center of your layout that reads ‘Hello World!’ Click on the text and hit ‘Delete’ to remove it from the view.
3. Still using the graphical editor, drag in a “Button” element from the left pane under the category “Form Widgets.” You will notice that crosshairs appear when the Button is centered in the view.



9. Select the button to edit its properties. On the right-hand side, edit the “ID” and “Text” properties as follows:
    a. ID: **@+id/beam_button** This identifier will be used by the activity to create a Button object in the Java code. The “@+id/” prefix instructs the compiler to create this as a new ID in a generated Java file of constants (named R.java); this prefix is required for the ID to be visible to Activities. You must save your XML files and build the project for the changes to be reflected in the generated file.
    b. Text: **Beam** The text property defines the text that the button displays to the user.



Editing an Activity

1. Open MainActivity.java to view the generated source code for the activity that has already been created

```
1  package com.tapped.nfc;
2
3  import android.app.Activity;
6
7  public class MainActivity extends Activity {
8
9      @Override
10     public void onCreate(Bundle savedInstanceState) {
11         super.onCreate(savedInstanceState);
12         setContentView(R.layout.activity_main);
13     }
14
15     @Override
16     public boolean onCreateOptionsMenu(Menu menu) {
17         getMenuInflater().inflate(R.menu.activity_main, menu);
18         return true;
19     }
20
21
22 }
23
```

2. Take note of the following in the screenshot above:
    c. MainActivity extends android.app.Activity. All activities must subclass the Android Activity class.
    d. The Activity is not launched with a main() method, but rather is initiated by the Android system by invoking a set of callbacks that correspond to specific stages of the lifecycle.
    e. Overriding the public method OnCreate() on line 10 allows us to specifically define what happens when the activity is starting. All derived classes of android.app.Activity must both implement OnCreate() as well as call the superclass's implementation (see line 11) of the method before invoking any additional code.
    f. Within OnCreate(), the Activity specifies the layout resource file that it wishes to display on line 12 with a call to setContentView(). Notice that the layout file you worked with earlier has already been generated by Eclipse as an accessible constant in in the R.java class.
    g. The onCreateOptionsMenu() method has also been generated and overridden in this activity. For now, it is safe to ignore this method; its utility will not be pertinent to this tutorial.

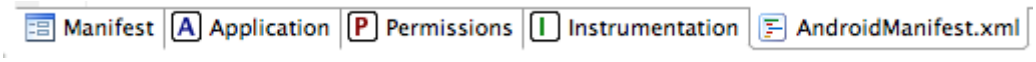
3. In the OnCreate() method, add the following block of code (from lines 17-25) to create an instance of a Button (a subclass of android.view.View) that will allow you to programmatically define its behavior.

```
12    @Override
13    public void onCreate(Bundle savedInstanceState) {
14        super.onCreate(savedInstanceState);
15        setContentView(R.layout.activity_main);
16
17        Button beamButton = (Button) findViewById(R.id.beam_button);
18        beamButton.setOnClickListener(new View.OnClickListener() {
19
20            @Override
21            public void onClick(View v) {
22                Toast.makeText(MainActivity.this, "Button Tapped!",
23                        Toast.LENGTH_LONG).show();
24            }
25        });
26    }
```

4. Take note of the following:
    h. The findViewById() method again takes advantage of the generated constant in R.java by referencing the ID defined earlier in *activity_main.xml*
    i. findViewById() returns a View which can be cast to a Button (since Button - and all UI elements defined in XML layout files - subclass android.view.View)
    j. Any clickable view (most are by default) allows for an anonymous listener to be set. In this case, a View.OnClickListener will define a callback for the OnClick() event. This particular Button will simply show a message to the user when tapped.

Making Sense of the Manifest

5. Open AndroidManifest.xml to reveal the Application's "Manifest" in the Eclipse editor. Notice that there are several tabs; select the XML editor for now. The markup is straightforward enough to understand and modify with little effort.



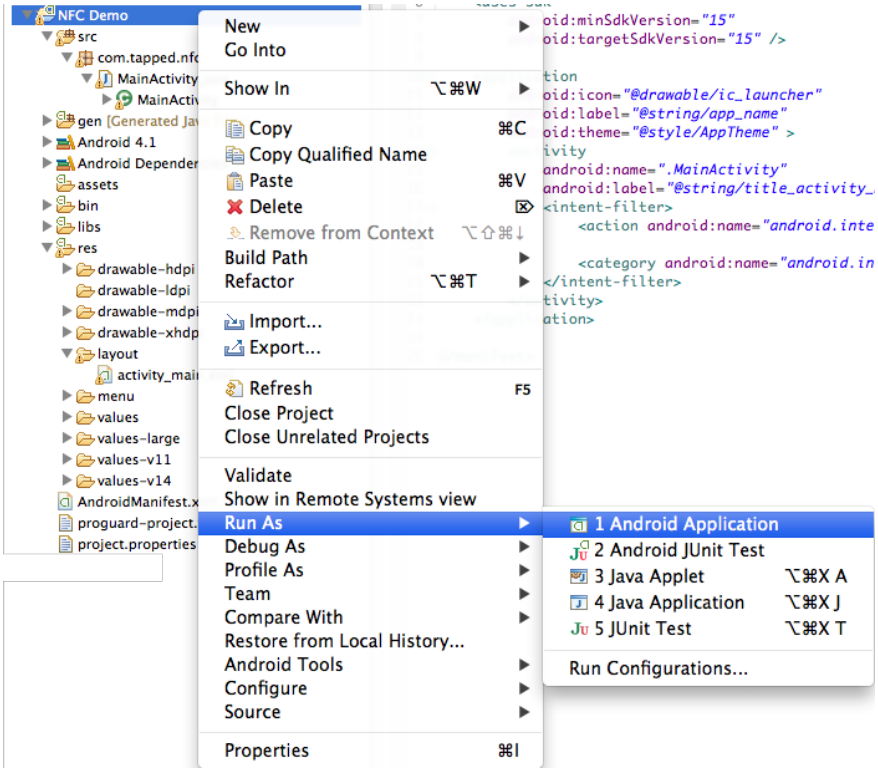
6. The Manifest defines all of the activities that are present in the application. Each time you create a new activity, you must add it to the manifest file, specifying its name and label attributes.

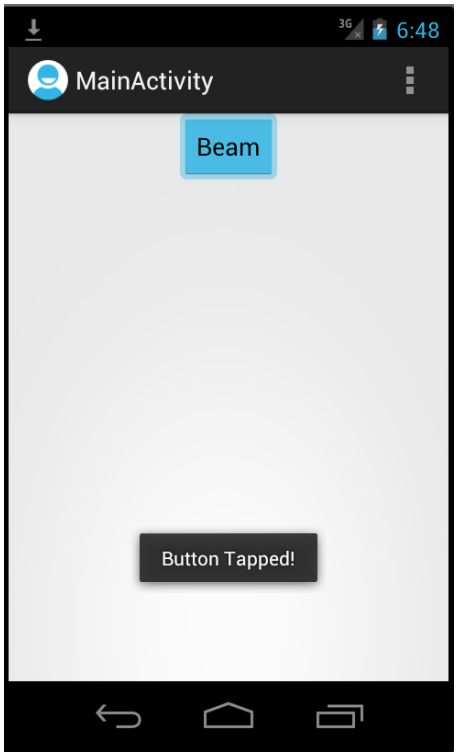
```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.tapped.nfc"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="15"
        android:targetSdkVersion="15" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="@string/title_activity_main" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

7. Notice that MainActivity defines a list of inner properties in a block named “Intent Filters”
    k. Intent filters are a way of defining which types of actions an Activity should respond to.
    l. The activity that is first created and displayed when your application is started must define intent.action.MAIN and intent.category.LAUNCHER in its intent filters.
    m. Intents and intent filtering (in the Manifest file) is particularly useful when defining how Activities respond to system NFC events.

(Aside) What is an Intent? Android’s documentation explains that intents are *“core components of an application … are activated through messages, called intents. Intent messaging is a facility for late run-time binding between components in the same or different applications. The intent itself, an Intent object, is a passive data structure holding an abstract description of an operation to be performed... In each case [of an intent], the Android system finds the appropriate activity... to respond to the intent, instantiating them if necessary.”*

8. At this point your application is ready to run - you can launch it to an AVD or Android device connected via USB simply by right-clicking on the root of the project in the left-hand pane and selecting Run > Run as Android Application


NFC Demo
src
com.tapped.nfc
MainActivity
gen [Generated Jav
Android 4.1
Android Depender
assets
bin
libs
res
drawable-hdpi
drawable-ldpi
drawable-mdpi
drawable-xhdp
layout
menu
values
values-large
values-v11
values-v14
AndroidManifest.x
proguard-project.
project.properties
New
Go Into
Show In ⌥⌘W
Copy ⌘C
Copy Qualified Name
Paste ⌘V
Delete
Remove from Context ⌥⇧⌘↓
Build Path
Refactor ⌥⌘T
Import...
Export...
Refresh F5
Close Project
Close Unrelated Projects
Validate
Show in Remote Systems view
Run As
Debug As
Profile As
Team
Compare With
Restore from Local History...
Android Tools
Configure
Source
Properties ⌘I
1 Android Application
2 Android JUnit Test
3 Java Applet ⌥⌘X A
4 Java Application ⌥⌘X J
5 JUnit Test ⌥⌘X T
Run Configurations...
oid:minSdkVersion="15"
oid:targetSdkVersion="15" />
tion
oid:icon="@drawable/ic_launcher"
oid:label="@string/app_name"
oid:theme="@style/AppTheme" >
ivity
android:name=".MainActivity"
android:label="@string/title_activity_
<intent-filter>
<action android:name="android.inte
<category android:name="android.in
</intent-filter>
tivity>
ation>


6:48
MainActivity
Beam
Button Tapped!

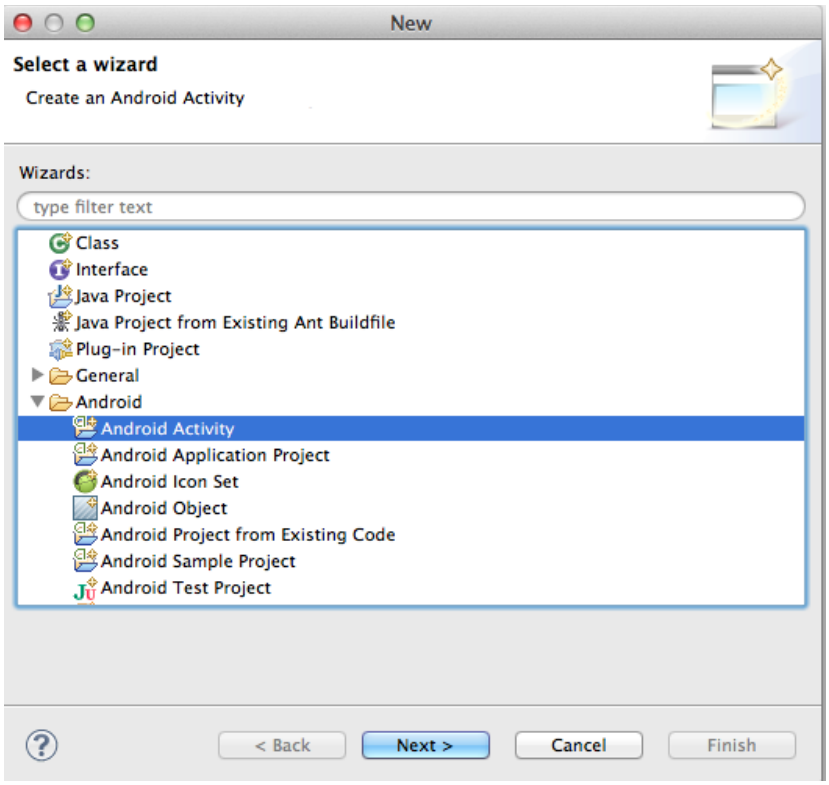
Adding Android Beam

Beam abstracts the Android NFC service layer into an easy-to-use protocol for peer-to-peer NFC communication. Google writes *"Android Beam allows simple peer-to-peer data exchange between two Android-powered devices. The application that wants to beam data to another device must be in the foreground and the device receiving the data must not be locked. When the beaming device comes in close enough contact with a receiving device, the beaming device displays the "Touch to Beam" UI. The user can then choose whether or not to beam the message to the receiving device."*

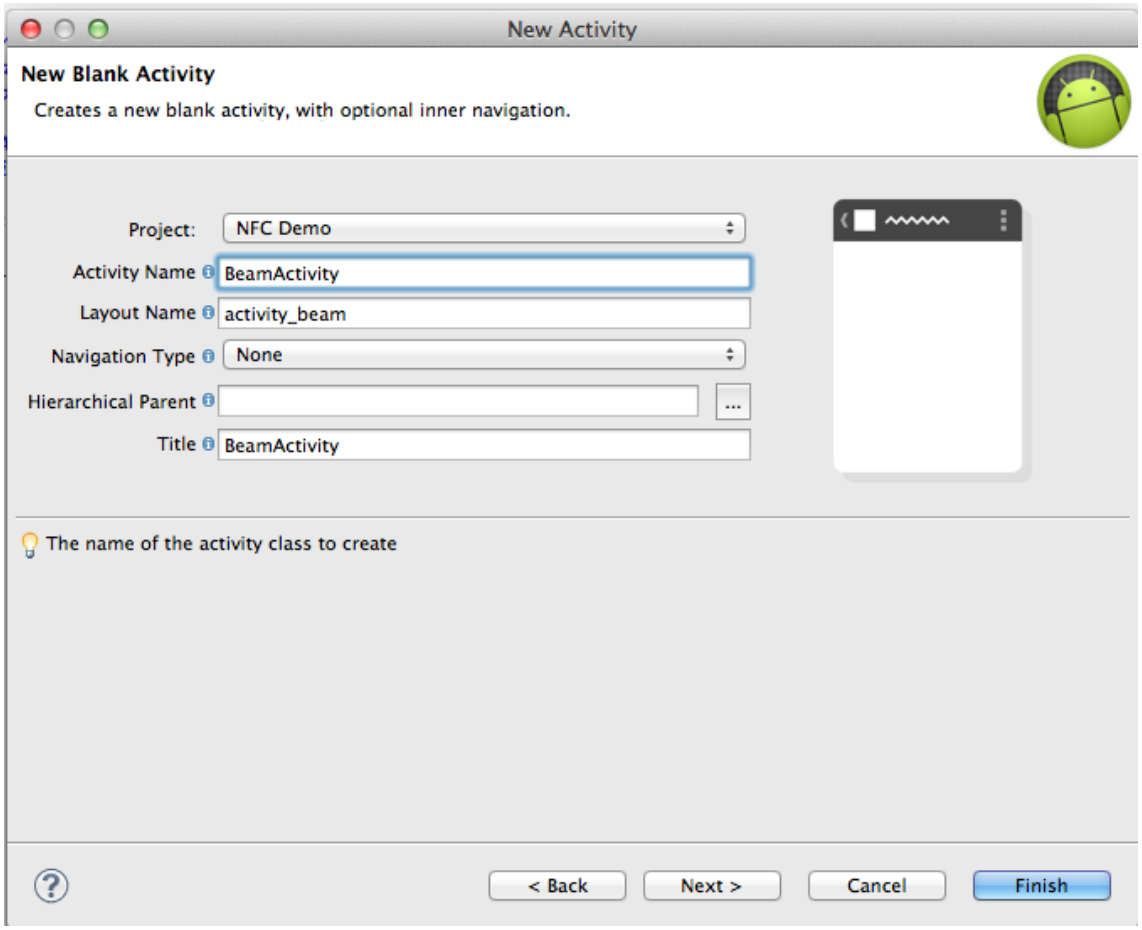
The goal of this section of the tutorial is to create a new activity in our existing project that will implement Android Beam. The activity will have a simple layout with a text field for the user to enter a string of text that will be beamed to the destination device.

Creating an Activity for Beam

1. On the left-hand side of your eclipse workspace, right click on the *src* directory and choose **New > Other**. In the dialog that opens, open the Android package and choose **"Android Activity"**



2. On the following screen, select 'BlankActivity' and click Next. The subsequent screen will allow you to name the activity. Type in 'BeamActivity' for the Activity Name and 'activity_beam' for the Layout Name.



3. The following screen will prompt you to refactor various project files as needed. Notice that the Android Manifest is automatically appended with a new entry for BeamActivity. Review the changes and select 'Finish' to complete the process.
4. In the newly created activity's layout, add the editable text field element EditText to the layout.
    a. In the layout editor, you can drag the item "Person Name" into your layout.
    b. Set the ID property to *beam_edit_text* such that the activity can reference the view element by *R.id.beam_edit_text*.
    c. Refer to the previous section for a review on linking UI elements from XML layouts into Activities.
5. In BeamActivity:
    a. Create member variables for an EditText and an NfcAdapter (Lines 16-17)
    b. In OnCreate, initialize the aforementioned variables (lines 24, 27)
    c. Set two callbacks on the NfcAdapter (lines 33,35)
        i. *NdefPushMessageCallback* (called when Beam has started)
        ii. *OnNdefPushCompleteCallback* (called when your device successfully pushes a beam message)

d. BeamActivity must implement *CreateNdefMessageCallback* and *OnNdefPushCompleteCallback* (Lines 14-15). The implementation of these methods will be discussed in greater detail in the following section.

```
14 public class BeamActivity extends Activity implements CreateNdefMessageCallback,
15         OnNdefPushCompleteCallback {
16     NfcAdapter mNfcAdapter;
17     EditText mEditText;
18
19     @Override
20     public void onCreate(Bundle savedInstanceState) {
21         super.onCreate(savedInstanceState);
22         setContentView(R.layout.activity_beam);
23
24         mEditText = (EditText) findViewById(R.id.beam_edit_text);
25
26         // Check for available NFC Adapter
27         mNfcAdapter = NfcAdapter.getDefaultAdapter(this);
28         if (mNfcAdapter == null) {
29             Toast.makeText(this, "Sorry, NFC is not available on this device",
30                     Toast.LENGTH_SHORT).show();
31         }
32         // Register callback to set NDEF message
33         mNfcAdapter.setNdefPushMessageCallback(this, this);
34         // Register callback to listen for message-sent success
35         mNfcAdapter.setOnNdefPushCompleteCallback(this, this);
36     }
```

Implementing Beam Callbacks

1. Implement the CreateNdefMessageCallback interface (see screenshot below)
    e. This callback returns an NdefMessage (which is the vessel for the information transferred over NFC) when your application is ready to transmit using Beam. It is important to understand that this code will not be executed until your device has initiated Beam with another device.
    f. In line 49, the String variable is initialized with the contents of the UI's text field
    g. An NdefMessage is constructed using a constructor that accepts an array of NdefRecords as its parameter
    h. The first NdefRecord is created using a utility class NfcUtils (provided in the source code of this project) which defines some helpful methods for creating and parsing NFC messages.
    i. The method *createRecord* used on line 51 creates an NdefRecord using a mime type and a byte array
        i. The mime type is a String defined as a constant in BeamActivity. This string defines a custom mime type specific to this application and is of the format *application/{package name}*. Each application should define this constant using the same format with its own unique package identifier.

```
private static final String MIME_TYPE = "application/com.tapped.nfc";
```

ii. The final parameter is the record's payload

j. The second NdefRecord that is used to construct the NdefMessage is an Android Application Record (AAR). The AAR is created using createApplicationRecord() which takes the package name (for example, *com.tapped.nfc*) as its parameter. Notice that this is a different string than the previously defined custom mime type.

```
44      /**
45       * Implementation for the CreateNdefMessageCallback interface
46       */
47      @Override
48      public NdefMessage createNdefMessage(NfcEvent event) {
49          String text = mEditText.getText().toString();
50          NdefMessage msg = new NdefMessage(new NdefRecord[] {
51                  NfcUtils.createRecord(MIME_TYPE, text.getBytes()),
52                  NdefRecord.createApplicationRecord(PACKAGE_NAME) });
53          return msg;
54      }
```

(Aside) What is an Android Application Record? Google states that *"an Android Application Record (AAR) provides a stronger certainty that your application is started when an NFC tag is scanned. An AAR has the package name of an application embedded inside an NDEF record. You can add an AAR to any NDEF record of your NDEF message, because Android searches the entire NDEF message for AARs. If it finds an AAR, it starts the application based on the package name inside the AAR. If the application is not present on the device, Google Play is launched to download the application."* Although you can add an AAR at any index in the array of NdefRecords, you should always add it as the last item; the system will not read any records once it encounters an AAR.

2. Next, implement the onNdefPushComplete callback interface
    k. Create a constant for the activity to use for messaging between threads (line 55)
    l. Define a handler to run an appropriate callback on the UI thread (line 58). Attempting to modify UI elements or run UI elements on any thread other than the UI thread will throw an exception.
    m. Override the onNdefPushComplete method and call the handler. This method returns an NfcEvent with the nfcAdapter field set, which isn't useful for our implementation

```
55      private static final int MESSAGE_SENT = 1;
56
57      /** This handler receives a message from onNdefPushComplete */
58      private final Handler mHandler = new Handler() {
59          @Override
60          public void handleMessage(Message msg) {
61              switch (msg.what) {
62              case MESSAGE_SENT:
63                  Toast.makeText(getApplicationContext(), "Message sent!",
64                          Toast.LENGTH_LONG).show();
65                  break;
66              }
67          }
68      };
69
70      /**
71       * Implementation for the OnNdefPushCompleteCallback interface
72       */
73      @Override
74      public void onNdefPushComplete(NfcEvent arg0) {
75          // A handler is needed to send messages to the activity when this
76          // callback occurs, because it happens from a binder thread
77          mHandler.obtainMessage(MESSAGE_SENT).sendToTarget();
78      }
```

3. At this point, our activity has implemented the necessary mechanisms to send an NdefMessage over Beam. For another device to react to the data sent, there are still additional steps to take.

Reacting to Beamed Data

1. Open the AndroidManifest in the XML editor view. You should see two activities within the application definition.
2. Add an intent filter in the definition of BeamActivity as shown below.
    a. The action *NDEF_DISCOVERED* instructs the system that this activity is a candidate to handle this action when generated elsewhere in the system. The Android system's NFC Service will generates this action when a Beam is received.
    b. Restricting the data with a custom mime type will further specify the types of intents that this activity should react to. In this case, using the mime type that was previously defined in the Beam messages will allow only this activity to receive its own Beams.
4. Finally, notice that launchMode has been set to *singleTop*. This prevents multiple instances of the activity from being created when a new intent attempts to start it.

```
27          <activity
28              android:name=".BeamActivity"
29              android:label="@string/title_activity_beam"
30              android:launchMode="singleTop" >
31              <intent-filter>
32                  <action android:name="android.nfc.action.NDEF_DISCOVERED" />
33                  <category android:name="android.intent.category.DEFAULT" />
34                  <data android:mimeType="application/com.tapped.nfc" />
35              </intent-filter>
36          </activity>
```

3. Return to the Java editor and view BeamActivity. There are two general states in which the activity will receive a new intent from the system's NFC service:
    a. The activity (and/or application) is not running, and must be created
    b. The activity is running and is displayed in the foreground
4. A simple way to deal with both cases using Android's lifecycle callback methods is to override both onNewIntent() and onResume().
    a. In onNewIntent(), on lines 84-87, we simply update the Activity's intent (a member variable). The onNewIntent() method is called for Activities that launch in *singleTop* mode.
    b. In onResume() on lines 90-96, parse the content of the intent's action to determine if the action should be processed for data sent over Beam.

```
83      @Override
84      public void onNewIntent(Intent intent) {
85          // onResume gets called after this to handle the intent
86          setIntent(intent);
87      }
88
89      @Override
90      public void onResume() {
91          super.onResume();
92          // Check to see that the Activity started due to an Android Beam
93          if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(getIntent().getAction())) {
94              processIntent(getIntent());
95          }
96      }
```

5. Previously, in the *CreateNdefMessage* callback, we defined an NdefMessage with two records; a payload record and an application record. The *processIntent()* method on lines 101 - 110 serves to parse and act on an intent generated by the NFC service for the action *NDEF_DISCOVERED* which will contain a single message in this case.
6. The payload read on line 107 holds the data our application is interested in - this contains the text that the EditText form field of the sender's device sent. Finally, on lines 108-109, display a toast to the user with the contents of the payload

```
98      /**
99       * Parses the NDEF Message from the intent and toast to the user
100      */
101     void processIntent(Intent intent) {
102         Parcelable[] rawMsgs = intent.getParcelableArrayExtra(
103                 NfcAdapter.EXTRA_NDEF_MESSAGES);
104         // in this context, only one message was sent over beam
105         NdefMessage msg = (NdefMessage) rawMsgs[0];
106         // record 0 contains the MIME type, record 1 is the AAR, if present
107         String payload = new String(msg.getRecords()[0].getPayload());
108         Toast.makeText(getApplicationContext(), "Message received over beam: " + payload,
109                 Toast.LENGTH_LONG).show();
110     }
```

Linking and Testing

7. **TODO**: **Screenshot:** Finally, in MainActivity, change the behavior of the Beam button to launch BeamActivity using an intent. This is essentially a navigation mechanism.
8. **TODO: Elaborate:** To test, you'll need two NFC-enabled Android devices. Remember to enable the NFC hardware in the device's settings screen.