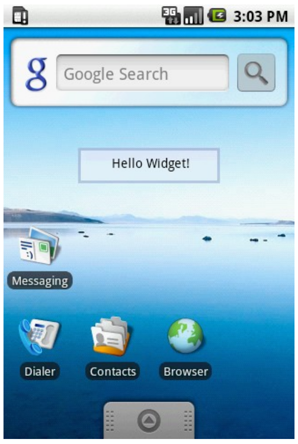
How to: Android 'Hello Widget'

Table of Contents

Document history

Version	Date	User	Description
1.0	2009-07-11	Norbert Möhring moehring.n [at] googlemail.com	Initial document.
1.1	2009-07-21	Norbert Möhring moehring.n [at] googlemail.com	Small fixes in code (copy&paste erros ☺). Thanks to Andreas Kompanez (ak@endlessnumbered.com) for the review. Better layout. Added document history and table of contents. Put all links behind text → looks better ☺
1.2	2009-07-23	Norbert Möhring moehring.n [at] googlemail.com	Fixed some typos
1.3	2009-09-27	Norbert Möhring moehring.n [at] googlemail.com	- Some comments to the new 1.6 Version - How to use Buttons on widgets

HelloWidget



Since there is only the one not that self explaining example of a widget I decided to invest some nightly hours to cut that example into pieces and then start from scratch with an easy to follow "Hello Widget" tutorial.

Prerequisites

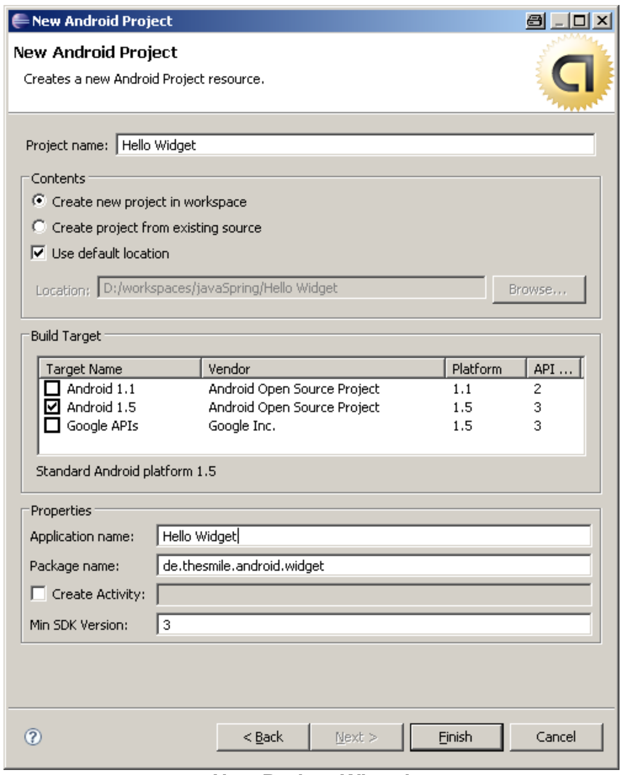
You should already have the android SDK and android Eclipse IDE plug-in installed and running. If not, go here to learn how to get started with the android plug-in. Also you should have at least basic knowledge about Java programming since this is not a Java tutorial.

Let's start from scratch

In Eclipse, go to

```
File → new Project … → other … and select 'Android Project'
```

The project name will be "Hello Widget" and in this case the target platform will be 'Android 1.5'. Uncheck the probably already checked Box "Create Activity". We won't create an Activity here we just want a simple widget.


New Android Project

Creates a new Android Project resource.

Project name: Hello Widget

Contents
- (●) Create new project in workspace
- ( ) Create project from existing source
- [✓] Use default location

Location: D:/workspaces/javaSpring/Hello Widget

Build Target

Target Name	Vendor	Platform	API ...
☐ Android 1.1	Android Open Source Project	1.1	2
☑ Android 1.5	Android Open Source Project	1.5	3
☐ Google APIs	Google Inc.	1.5	3

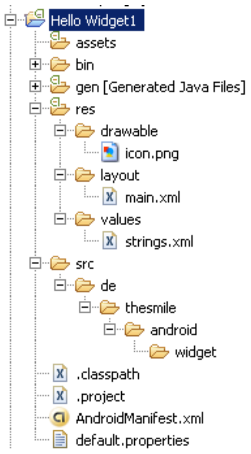
Standard Android platform 1.5

Properties

Application name:	Hello Widget
Package name:	de.thesmile.android.widget
☐ Create Activity:	
Min SDK Version:	3

New Project Wizard

After that, your Project structure will look like this:


```
Hello Widget1
  assets
  bin
  gen [Generated Java Files]
  res
    drawable
      icon.png
    layout
      main.xml
    values
      strings.xml
  src
    de
      thesmile
        android
          widget
  .classpath
  .project
  AndroidManifest.xml
  default.properties
```

The project wizard gave us some default stuff already, like the default android app-icon e.g.
We’ll start with the layout and design of our widget.
Open *main.xml* and modify it like this:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:orientation="vertical"
   android:background="@drawable/widget_bg_normal"
   android:layout_gravity="center"
   android:layout_height="wrap_content">

<TextView android:id="@+id/widget_textview"
   android:text="@string/widget_text"
   android:layout_height="wrap_content"
   android:layout_width="wrap_content"
   android:layout_gravity="center_horizontal|center"
   android:layout_marginTop="5dip"
   android:padding="10dip"
   android:textColor="@android:color/black"/>

</LinearLayout>
```

We have a simple linear layout with a TextView for your message to display. At this point, you will get an Error saying

```
ERROR Error: No resource found that matches the given name (at 'background'
with value '@drawable/widget_bg_normal')
```

That's because you don't have the **widget_bg_normal** resource at this point. I borrowed the image from the SimpleWiktionary widget. It's a png NinePatch image which will be our background of the widget.
Put the file *widget_bg_normal.9.png* (or your own background image) into the folder *res/drawable*.
You also don't have the *@string/widget_text* resource. To fix that, go to *res/values/string.xml* and add this line to it:

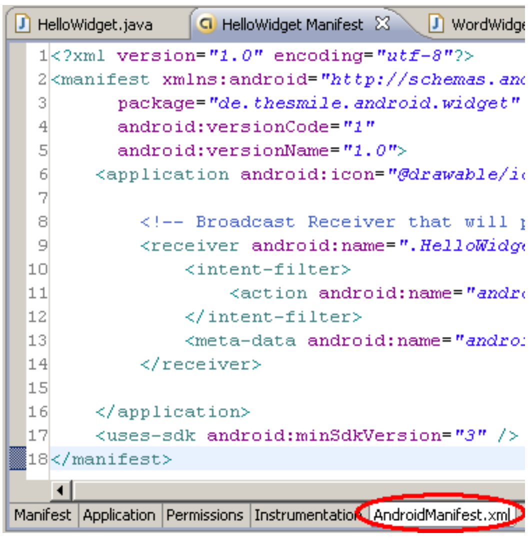
```
<string name="widget_text">Hello Widget!</string>
```

string.xml will look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="widget_text">Hello Widget!</string>
    <string name="app_name">Hello Widget</string>
</resources>
```

We just finished our design part.

Now we have to tell Android that we are going to have a widget.
Open *AndroidManifest.xml* and go to the source view.



```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.anc
      package="de.thesmile.android.widget"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:icon="@drawable/ic

        <!-- Broadcast Receiver that will p
        <receiver android:name=".HelloWidge
            <intent-filter>
                <action android:name="andr
            </intent-filter>
            <meta-data android:name="andro:
        </receiver>

    </application>
    <uses-sdk android:minSdkVersion="3" />
</manifest>
```

The lines you have to add are bold:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="de.thesmile.android.widget"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">

        <!-- Broadcast Receiver that will process AppWidget updates -->
        <receiver android:name=".HelloWidget" android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
            </intent-filter>
            <meta-data android:name="android.appwidget.provider"
                                  android:resource="@xml/hello_widget_provider" />
        </receiver>

    </application>
    <uses-sdk android:minSdkVersion="3" />
</manifest>
```

We declare our Broadcast Receiver which will be “HelloWidget”.
Don’t forget the dot before HelloWidget, this will create the full qualified path name with the declared package at the top of the file. The Label of our application was already set be the project wizard and is located in our *string.xml.*

From the documentation:
The <intent-filter> element must include an <action> element with the android:name attribute. This attribute specifies that the AppWidgetProvider accepts the ACTION_APPWIDGET_UPDATE broadcast. This is the only broadcast that you must explicitly declare. The AppWidgetManager automatically sends all other App Widget broadcasts to the AppWidgetProvider as necessary.

The meta-tag tells android about your widget provider. In this case, the widget provider is located at *res/xml/hello_widget_provider.xml*. The provider.xml is pretty much self explaining:

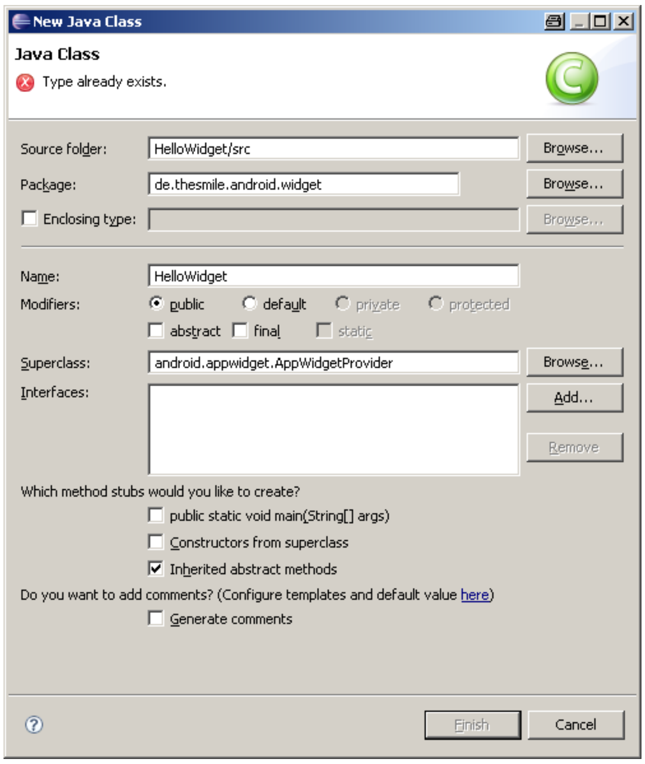
```
<?xml version="1.0" encoding="utf-8"?>
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
    android:minWidth="146dip"
    android:minHeight="72dip"
    android:updatePeriodMillis="10000"
    android:initialLayout="@layout/main"
/>
```

Define the size of your widget (should match the guidelines for desktop widgets).
updatePerdiodMillis as it’s name already says, the time in milliseconds to update your widget. In our case, this is unimportant and we don’t need it because we don’t do any update on our widget.

The initial layout points the the *main.xml* file. We already defined our layout and design.

The only thing that is missing to run our widget is the Class that extends the `AppWidgetProvider` which we declared in *AndroidManifest.xml* at the `receiver`-tag.

Create a new Class under the package `de.thesmile.android.widget`, name it `HelloWidget` and set `AppWidgetProvider` as super class.



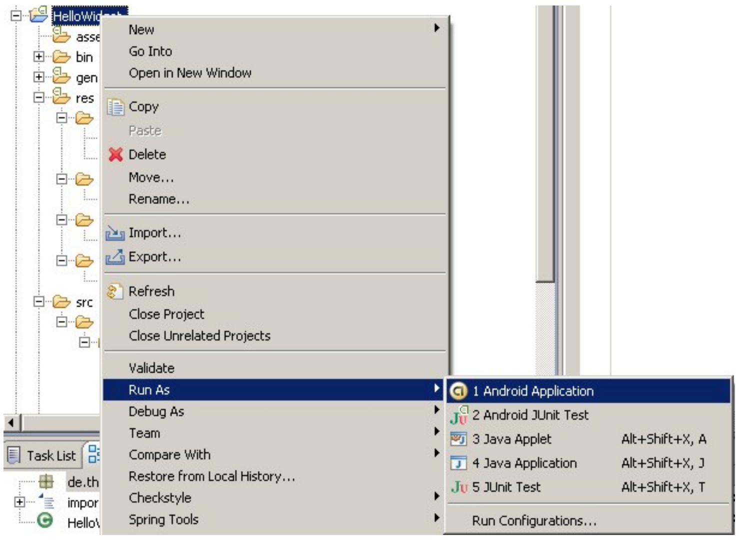
Type already exists.

Source folder: HelloWidget/src
Package: de.thesmile.android.widget
Enclosing type: [ ]
Name: HelloWidget
Modifiers: [✓ public] default private protected
abstract final static
Superclass: android.appwidget.AppWidgetProvider
Interfaces:
Which method stubs would you like to create?
[ ] public static void main(String[] args)
[ ] Constructors from superclass
[✓] Inherited abstract methods
Do you want to add comments? (Configure templates and default value here)
[ ] Generate comments

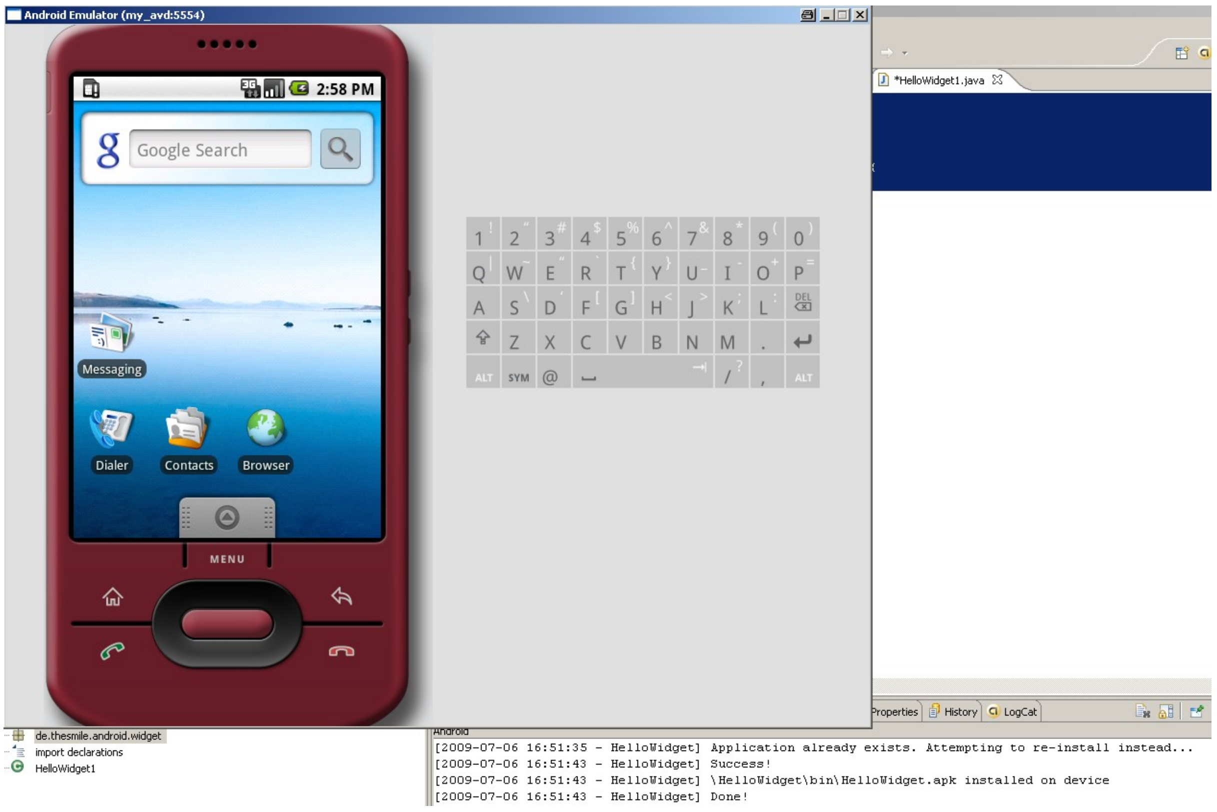
Your new class will look like this:

```
package de.thesmile.android.widget;

import android.appwidget.AppWidgetProvider;

public class HelloWidget extends AppWidgetProvider {

}
```

Since our Widget is doing nothing, you're now ready to go. To start your application, right click on your project → `Run As` → `Android Application`

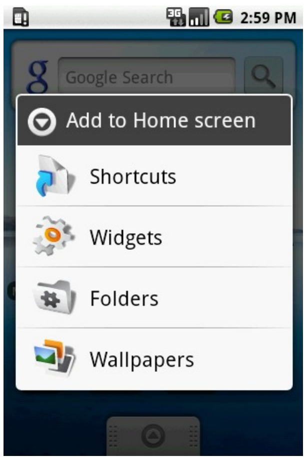


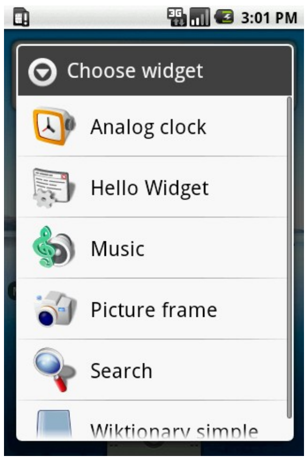
This will start the Android Virtual Device (AVD) and install your widget on this device.



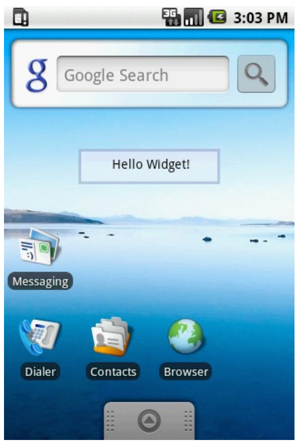
```
[2009-07-06 16:51:35 - HelloWidget] Application already exists. Attempting to re-install instead...
[2009-07-06 16:51:43 - HelloWidget] Success!
[2009-07-06 16:51:43 - HelloWidget] \HelloWidget\bin\HelloWidget.apk installed on device
[2009-07-06 16:51:43 - HelloWidget] Done!
```

Now press and hold your left mouse button on the AVD Desktop screen until the menu pops up. Select `Widgets → Hello Widget` …



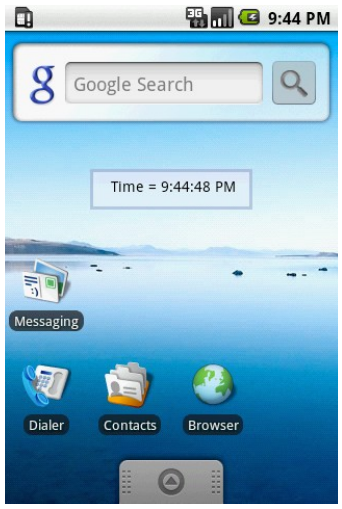


… and have a look at the result.



Ok, that was pretty easy and now you have a widget which can do …. nothing ☺

Let's do something with our widget ... display the current time. I'll show you two versions of the time example, one that I would call the *Java-way* and the other one to introduce you to services in widgets (*Service-way*).



The *Java-way* Time-Widget

This is how the Java-way Time-Widget will look like. It will update every second and display the time.

By the way: It's 11:50 pm right now, I don't know where the AVD gets it's time from but it's definitely not my system time.

Now, open your empty class `HelloWidget`.
We have to override the protected method `onUpdate` from `AppWidgetProvider`. Also, because of the Java-way, we'll create an inner class MyTime which will be derived from TimerTask. Just have a look at the following code:

```
public class HelloWidget extends AppWidgetProvider {

        @Override
        public void onUpdate(Context context, AppWidgetManager appWidgetManager,
                                                       int[] appWidgetIds) {

                Timer timer = new Timer();
                timer.scheduleAtFixedRate(new MyTime(context, appWidgetManager), 1, 1000);
        }

        private class MyTime extends TimerTask {

                RemoteViews remoteViews;
                AppWidgetManager appWidgetManager;
                ComponentName thisWidget;
                DateFormat format = SimpleDateFormat.getTimeInstance(SimpleDateFormat.MEDIUM,
                                        Locale.getDefault());

                public MyTime(Context context, AppWidgetManager appWidgetManager) {
                        this.appWidgetManager = appWidgetManager;
                        remoteViews = new RemoteViews(context.getPackageName(), R.layout.main);
                        thisWidget = new ComponentName(context, HelloWidget.class);
                }

                @Override
                public void run() {
                        remoteViews.setTextViewText(R.id.widget_textview,
                                                        "Time = " + format.format(new Date()));
                        appWidgetManager.updateAppWidget(thisWidget, remoteViews);
                }
        }
}
```

The method `onUpdate` will be called at first. We create a timer that will run our MyTime-Thread every second. The constructor of the MyTime-Class gets some information to update our widget on the desktop.
Get our main view that we created (at that point you could also change your layout to another design).

```
        remoteViews = new RemoteViews(context.getPackageName(), R.layout.main);
```

In the run method, we set the time with our new Date and update the remote View.
That's already it. Now you have a widget that is displaying the current time.
At that point I want to give you some further information about a bug that has not been fixed yet, either in the AVD nor on any machine. The bug-fix didn't make it into the CUPCAKE Version (1.5).

To delete a widget, you tab and hold, then drag it and drop it into the trash can at the bottom.
At that point the widget provider usually runs an onDelete and onDisable method. Currently this is not happening since the wrong IDs are sent. To fix this problem and really delete your widget, add the following code to the onReceive method:

```
        @Override
        public void onReceive(Context context, Intent intent) {

                // v1.5 fix that doesn't call onDelete Action
                final String action = intent.getAction();
                if (AppWidgetManager.ACTION_APPWIDGET_DELETED.equals(action)) {
                        final int appWidgetId = intent.getExtras().getInt(
                                        AppWidgetManager.EXTRA_APPWIDGET_ID,
                                        AppWidgetManager.INVALID_APPWIDGET_ID);
                        if (appWidgetId != AppWidgetManager.INVALID_APPWIDGET_ID) {
                                this.onDeleted(context, new int[] { appWidgetId });
                        }
                } else {
                        super.onReceive(context, intent);
                }
        }
```

`onReceive` is always called before `onUpdate` is called. This code snipped checks the called action and decides weather to just run the receive method or the delete method.
Jeff Sharkey provided us with this useful code-snipped.

If you don't do that, your widget won't be displayed any more, but your widget provider will still run onReceive and onUpdate in the given period (updatePerdiodMillis) which will eat up battery and memory. It seems like this problem has been fixed in the 1.6 update so that you don't need this additional code anymore.

The ***Service-way*** **Time-Widget** *(not working with 1.6 any more)*

Usually your widget will do some more stuff than updating your time. To do so, you should create an `UpdateService.class`. Here's the Time-Widget with a service. This is just an example and I'm pretty sure that the service-way of the time widget will eat up more battery than the Java-way.

To register and start the service, change your onUpdate method like this:

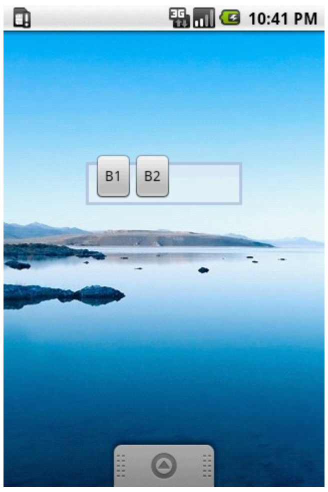
```
@Override
public void onUpdate(Context context, AppWidgetManager appWidgetManager,
              int[] appWidgetIds) {

       Intent intent = new Intent(context, UpdateService.class);
       context.startService(intent);
}
```

The UpdateService.class looks like this:

```
public static class UpdateService extends Service {

@Override
public void onStart(Intent intent, int startId) {
       RemoteViews updateViews = new RemoteViews(this.getPackageName(),
                                                 R.layout.main);
       Date date = new Date();
       DateFormat format = SimpleDateFormat.getTimeInstance(
                           SimpleDateFormat.MEDIUM, Locale.getDefault());

       // set the text of component TextView with id 'message'
       updateViews.setTextViewText(R.id.widget_textview, "Current Time "
                           + format.format(date));

       // Push update for this widget to the home screen
       ComponentName thisWidget = new ComponentName(this, HelloWidget.class);
       AppWidgetManager manager = AppWidgetManager.getInstance(this);
       manager.updateAppWidget(thisWidget, updateViews);
}
```

If you want the time to be updated every second, your have to modify the *hello_widget_provider.xml* file and set updatePerdiodMillis to 1000.
The android update 1.6 unfortunately does not support any update periode that is less then 30 minutes. The result is, that you can not run the “service-way” in 1.6 any more. Since updating every second is not very useful concerning the battery life of your phone, the 30 minute minimum period makes sense. Still, it is not very useful concerning development and testing because no developer wants to wait for 30 minutes to test the update of his widget.
Anyway, for usability purposes one should always use an AlarmManager for widget updates, since you can let the user configure the periode. I'll get to the AlarmManager in my next tutorial update.

How to use Buttons on Widgets



After a few requests about updating this tutorial to show how buttons work on widgets, I finally made it and here is an example App for buttons. This application will not be functional like the Time-Widget. It will just show you how you can add several buttons and click events.

Prerequisites

For the prerequisites, you already know how to create a widget project from scratch. Just do that, or download the source code for this example widget. You can get it here.

The Layout

For the layout, we use the *main.xml* which will look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
       android:layout_width="fill_parent"
       android:orientation="horizontal"
       android:background="@drawable/widget_bg_normal"
       android:layout_gravity="center"
       android:layout_height="wrap_content">
<Button android:text="B1" android:id="@+id/button_one"
       android:layout_gravity="center_horizontal|center"
       android:layout_height="fill_parent"
       android:layout_width="wrap_content"/>

<Button android:text="B2" android:id="@+id/button_two"
       android:layout_gravity="center_horizontal|center"
       android:layout_height="fill_parent"
       android:layout_width="wrap_content"/>
</LinearLayout>
```

We use the background from our first widget and add two buttons to it, that's all we need.
Since this example will also start an activity from a button, we have to create a layout for this activity. This e.g. could be a configuration screen. The *configure.xml* file looks like this:

```
<TableLayout android:id="@+id/TableLayout01"
              android:layout_width="fill_parent"
              android:layout_height="fill_parent"
              xmlns:android="http://schemas.android.com/apk/res/android">
       <TextView android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:id="@+id/TextOne"
              android:text="Text"
              android:textColor="@android:color/white"
              android:textSize="15dip"
              android:textStyle="bold"
              android:layout_marginTop="5dip"/>
       <EditText android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:id="@+id/txtFieldId"
              android:clickable="true"
              android:inputType="number"
              android:layout_marginTop="5dip"
              android:layout_marginBottom="20dip"/>
</TableLayout>
```

We just added a label with the string “Text” and a textfield to the screen. Nothing will be functional here.

AppWidget Provider

The appwidget-provider xml named *button_widget_provider.xml* is very simple. We don't need any update periode so it will look like this:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
    android:minWidth="146dip"
    android:minHeight="72dip"
    android:updatePeriodMillis="0"
    android:initialLayout="@layout/main"
/>
```

Manifest file

The manifest file will have some more information this time and should look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="de.thesmile.android.widget.buttons"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">

       <!-- Broadcast Receiver that will process AppWidget updates -->
        <receiver android:name=".ButtonWidget" android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
                <!-- Broadcast Receiver that will also process our self created action -->
               <action
android:name="de.thesmile.android.widget.buttons.ButtonWidget.ACTION_WIDGET_RECEIVER"/>
            </intent-filter>
            <meta-data android:name="android.appwidget.provider"
android:resource="@xml/button_widget_provider" />
        </receiver>

<!-- this activity will be called, when we fire our self created ACTION_WIDGET_CONFIGURE -->
               <activity android:name=".ClickOneActivity">
                       <intent-filter>
                       <action
android:name="de.thesmile.android.widget.buttons.ButtonWidget.ACTION_WIDGET_CONFIGURE"/>
                       </intent-filter>
               </activity>
    </application>
    <uses-sdk android:minSdkVersion="3" />
</manifest>
```

Ok, let's have a detailled look at it.
We already know about the first part. We register our ButtonWidget as a receiver which will react on the intent with the filter `APPWIDGET_UPDATE`. We also know about the meta-data tag which declares some widget properties. The new thing here is the `ACTION_WIDGET_RECEIVER`, which, as you can see, belongs to our package. What we do is, we add a certain event to our ButtonWidget-Receiver. If this event (android calls these events *intent*) is fired, our ButtonWidget will be called to react on it.

The second thing is the activity *ClickOneActivity.* We register the intent-filter `ACTION_WIDGET_CONFIGURE` to this activity.

The code for the ClickOneActivity is pretty straight forward and you already know how that is working. We only use the onCreate method here and display a Toastmessage:

```
public class ClickOneActivity extends Activity {
 @Override
 protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // change to our configure view
        setContentView(R.layout.configure);

        // don't call 'this', use 'getApplicationContext()', the activity-object is
        // bigger than just the context because the activity also stores the UI elemtents
        Toast.makeText(getApplicationContext(), "We are in ClickOneActivity",
               Toast.LENGTH_SHORT).show();
 }
}
```

Now we can have a look at the `AppWidgetProvider` code to see how the intent-filters work.

ButtonWidget Provider

First of all, we configure our own events which we already used in our manifest file. You should always make sure that your actions have unique names not only in the namespace of your own apps but in the global namespace of the whole android framework. My idea is to just use your package name of the application to insure that. (I didn't do that in this example)

```
public class ButtonWidget extends AppWidgetProvider {

	public static String ACTION_WIDGET_CONFIGURE = "ConfigureWidget";
	public static String ACTION_WIDGET_RECEIVER = "ActionReceiverWidget";

	[...]
```

Now, let's have a look at the most important method; `onUpdate()`

```
@Override
public void onUpdate(Context context, AppWidgetManager appWidgetManager, int[] appWidgetIds) {

 RemoteViews remoteViews = new RemoteViews(context.getPackageName(), R.layout.main);
 Intent configIntent = new Intent(context, ClickOneActivity.class);
 configIntent.setAction(ACTION_WIDGET_CONFIGURE);

 Intent active = new Intent(context, ButtonWidget.class);
 active.setAction(ACTION_WIDGET_RECEIVER);
 active.putExtra("msg", "Message for Button 1");

 PendingIntent actionPendingIntent = PendingIntent.getBroadcast(context, 0, active, 0);
 PendingIntent configPendingIntent = PendingIntent.getActivity(context, 0, configIntent, 0);

 remoteViews.setOnClickPendingIntent(R.id.button_one, actionPendingIntent);
 remoteViews.setOnClickPendingIntent(R.id.button_two, configPendingIntent);

 appWidgetManager.updateAppWidget(appWidgetIds, remoteViews);
}
```

First, we get the RemoteViews for our main layout; that's where we will be working on. Then we configure a new Intent that will hold our configuration activity. This event will be given our action *`ACTION_WIDGET_CONFIGURE`*.

The second Intent will be configured with our ButtonWidget and be given the action *`ACTION_WIDGET_RECEIVER`*. We also give this action some extra data that we will use later to display something on the screen. This is just to show you how you can add extra data to an intent and use it later when this intent is captured by whatever application is catching it.

Now we use PendingIntents which will be given to the remoteViews as click-events.

From the documentation:
By giving a PendingIntent to another application, you are granting it the right to perform the operation you have specified as if the other application was yourself (with the same permissions and identity).

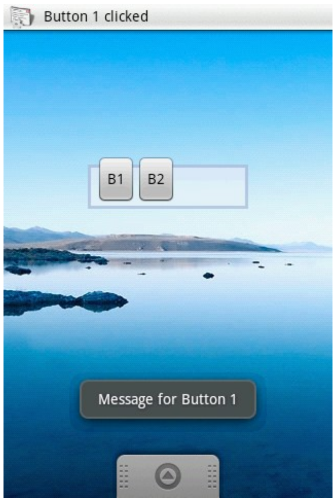
As usual, we have to update our widget with the `appWidgetManager` to „commit“ the updates. Now everything is registered to our widget and the button „B2“ will be working already since it will start our Activity as soon as you click the button.
Button one will do nothing at the moment because we didn't tell our ButtonWidget to do anything when it receives the action *`ACTION_WIDGET_RECEIVER`*.

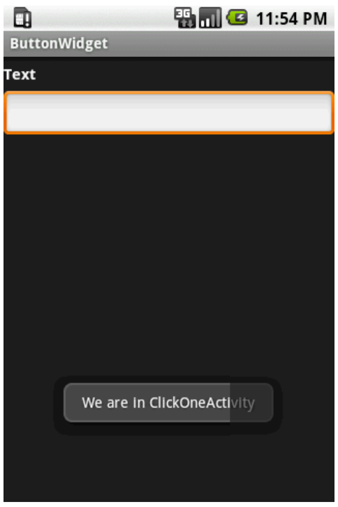
Let's have a look at the important method that can react on the *ACTION_WIDGET_RECEIVER*:

```
@Override
public void onReceive(Context context, Intent intent) {

// v1.5 fix that doesn't call onDelete Action
final String action = intent.getAction();
if (AppWidgetManager.ACTION_APPWIDGET_DELETED.equals(action)) {
        final int appWidgetId = intent.getExtras().getInt(
                AppWidgetManager.EXTRA_APPWIDGET_ID,
                AppWidgetManager.INVALID_APPWIDGET_ID);
        if (appWidgetId != AppWidgetManager.INVALID_APPWIDGET_ID) {
                this.onDeleted(context, new int[] { appWidgetId });
        }
} else {
  // check, if our Action was called
  if (intent.getAction().equals(ACTION_WIDGET_RECEIVER)) {
                String msg = "null";
                try {
                        msg = intent.getStringExtra("msg");
                } catch (NullPointerException e) {
                        Log.e("Error", "msg = null");
                }
                Toast.makeText(context, msg, Toast.LENGTH_SHORT).show();

        PendingIntent contentIntent = PendingIntent.getActivity(context, 0, intent, 0);
        NotificationManager notificationManager =
                (NotificationManager)context.getSystemService(Context.NOTIFICATION_SERVICE);
        Notification noty = new Notification(R.drawable.icon, "Button 1 clicked",
                System.currentTimeMillis());

        noty.setLatestEventInfo(context, "Notice", msg, contentIntent);
        notificationManager.notify(1, noty);

  }
  super.onReceive(context, intent);
 }
}
```

At the beginning I put the 1.5 onReceive-fix here, but this time we ask for our action in the else statement before we call super.onReceive. If *ACTION_WIDGET_RECEIVER* is called, we get our extra data from the intent (which was stored under the key „msg“) and display a Toast-Message with this string to the user.

As a little gimmick I also show you how you use the NotificationManager of the android framework. The message „Button 1 clicked“ will pop up on the notification bar (the bar where your clock is placed) and will also show the widget icon.





Feedback appreciated

Author: Norbert Möhring (moehring.n [at] googlemail.com)
Blog: http://blog.thesmile.de