23 September 2009

An introduction to Text-To-Speech in Android

We've introduced a new feature in version 1.6 of the Android platform: Text-To-Speech (TTS). Also known as "speech synthesis", TTS enables your Android device to "speak" text of different languages.

Before we explain how to use the TTS API itself, let's first review a few aspects of the engine that will be important to your TTS-enabled application. We will then show how to make your Android application talk and how to configure the way it speaks.

Languages and resources

About the TTS resources

The TTS engine that ships with the Android platform supports a number of languages: English, French, German, Italian and Spanish. Also, depending on which side of the Atlantic you are on, American and British accents for English are both supported.

The TTS engine needs to know which language to speak, as a word like "Paris", for example, is pronounced differently in French and English. So the voice and

dictionary are language-specific resources that need to be loaded before the engine can start to speak.

Although all Android-powered devices that support the TTS functionality ship with the engine, some devices have limited storage and may lack the language-specific resource files. If a user wants to install those resources, the TTS API enables an application to query the platform for the availability of language files and can initiate their download and installation. So upon creating your activity, a good first step is to check for the presence of the TTS resources with the corresponding intent:

```
Intent checkIntent = new Intent();
checkIntent.setAction(TextToSpeech.Engine.ACTION_CHECK_TTS_DATA);
startActivityForResult(checkIntent, MY_DATA_CHECK_CODE);
```

A successful check will be marked by a `CHECK_VOICE_DATA_PASS` result code, indicating this device is ready to speak, after the creation of our `android.speech.tts.TextToSpeech` object. If not, we need to let the user know to install the data that's required for the device to become a multi-lingual talking machine! Downloading and installing the data is accomplished by firing off the ACTION_INSTALL_TTS_DATA intent, which will take the user to Android Market, and will let her/him initiate the download. Installation of the data will happen automatically once the download completes. Here is an example of what your implementation of `onActivityResult()` would look like:

```
private TextToSpeech mTts;
protected void onActivityResult(
        int requestCode, int resultCode, Intent data) {
    if (requestCode == MY_DATA_CHECK_CODE) {
        if (resultCode == TextToSpeech.Engine.CHECK_VOICE_DATA_PASS) {
            // success, create the TTS instance
            mTts = new TextToSpeech(this, this);
        } else {
            // missing data, install it
            Intent installIntent = new Intent();
            installIntent.setAction(
                TextToSpeech.Engine.ACTION_INSTALL_TTS_DATA);
            startActivity(installIntent);
        }
    }
}
```

In the constructor of the `TextToSpeech` instance we pass a reference to the `Context` to be used (here the current Activity), and to an `OnInitListener` (here our Activity as well). This listener enables our application to be notified when the Text-To-Speech engine is fully loaded, so we can start configuring it and using it.

Languages and Locale

At Google I/O, we showed an example of TTS where it was used to speak the result of a translation from and to one of the 5 languages the Android TTS engine currently supports. Loading a language is as simple as calling for instance:

```
mTts.setLanguage(Locale.US);
```

to load and set the language to English, as spoken in the country "US". A locale is the preferred way to specify a language because it accounts for the fact that the same language can vary from one country to another. To query whether a specific Locale is supported, you can use isLanguageAvailable(), which returns the level of support for the given Locale. For instance the calls:

```
mTts.isLanguageAvailable(Locale.UK))
mTts.isLanguageAvailable(Locale.FRANCE))
mTts.isLanguageAvailable(new Locale("spa", "ESP")))
```

will return TextToSpeech.LANG_COUNTRY_AVAILABLE to indicate that the language AND country as described by the Locale parameter are supported (and the data is correctly installed). But the calls:

```
mTts.isLanguageAvailable(Locale.CANADA_FRENCH))
mTts.isLanguageAvailable(new Locale("spa"))
```

will return `TextToSpeech.LANG_AVAILABLE`. In the first example, French is supported, but not the given country. And in the second, only the language was specified for the Locale, so that's what the match was made on.

Also note that besides the `ACTION_CHECK_TTS_DATA` intent to check the availability of the TTS data, you can also use `isLanguageAvailable()` once you have created your `TextToSpeech` instance, which will return `TextToSpeech.LANG_MISSING_DATA` if the required resources are not installed for the queried language.

Making the engine speak an Italian string while the engine is set to the French language will produce some pretty *interesting* results, but it will not exactly be something your user would understand So try to match the language of your application's content and the language that you loaded in your `TextToSpeech` instance. Also if you are using `Locale.getDefault()` to query the current Locale, make sure that at least the default language is supported.

Making your application speak

Now that our `TextToSpeech` instance is properly initialized and configured, we can start to make your application speak. The simplest way to do so is to use the `speak()` method. Let's iterate on the following example to make a talking alarm clock:

```
String myText1 = "Did you sleep well?";
String myText2 = "I hope so, because it's time to wake up.";
mTts.speak(myText1, TextToSpeech.QUEUE_FLUSH, null);
mTts.speak(myText2, TextToSpeech.QUEUE_ADD, null);
```

The TTS engine manages a global queue of all the entries to synthesize, which are also known as "utterances". Each `TextToSpeech` instance can manage its own queue in order to control which utterance will interrupt the current one and which one is simply queued. Here the first `speak()` request would interrupt whatever was currently being synthesized: the queue is flushed and the new utterance is queued, which places it at the head of the queue. The second utterance is queued and will be played after `myText1` has completed.

Using optional parameters to change the playback stream type

On Android, each audio stream that is played is associated with one stream type, as defined in android.media.AudioManager. For a talking alarm clock, we would like our text to be played on the AudioManager.STREAM_ALARM stream type so that it respects the alarm settings the user has chosen on the device. The last parameter of the speak() method allows you to pass to the TTS engine optional parameters, specified as key/value pairs in a HashMap. Let's use that mechanism to change the stream type of our utterances:

```
HashMap<String, String> myHashAlarm = new HashMap();
myHashAlarm.put(TextToSpeech.Engine.KEY_PARAM_STREAM,
        String.valueOf(AudioManager.STREAM_ALARM));
mTts.speak(myText1, TextToSpeech.QUEUE_FLUSH, myHashAlarm);
mTts.speak(myText2, TextToSpeech.QUEUE_ADD, myHashAlarm);
```

Using optional parameters for playback completion callbacks

Note that `speak()` calls are asynchronous, so they will return well before the text is done being synthesized and played by Android, regardless of the use of `QUEUE_FLUSH` or `QUEUE_ADD`. But you might need to know when a particular utterance is done playing. For instance you might want to start playing an annoying music after `myText2` has finished synthesizing (remember, we're trying to wake up the user). We will again use an optional parameter, this time to tag our utterance as one we want to identify. We also need to make sure our activity implements the `TextToSpeech.OnUtteranceCompletedListener` interface:

```
mTts.setOnUtteranceCompletedListener(this);
myHashAlarm.put(TextToSpeech.Engine.KEY_PARAM_STREAM,
        String.valueOf(AudioManager.STREAM_ALARM));
mTts.speak(myText1, TextToSpeech.QUEUE_FLUSH, myHashAlarm);
myHashAlarm.put(TextToSpeech.Engine.KEY_PARAM_UTTERANCE_ID,
        "end of wakeup message ID");
// myHashAlarm now contains two optional parameters
mTts.speak(myText2, TextToSpeech.QUEUE_ADD, myHashAlarm);
```

And the Activity gets notified of the completion in the implementation of the listener:

```
public void onUtteranceCompleted(String uttId) {
    if (uttId == "end of wakeup message ID") {
        playAnnoyingMusic();
    }
}
```

File rendering and playback

While the `speak()` method is used to make Android speak the text right away, there are cases where you would want the result of the synthesis to be recorded in an audio file instead. This would be the case if, for instance, there is text your application will speak often; you could avoid the synthesis CPU-overhead by rendering only once to a file, and then playing back that audio file whenever needed. Just like for `speak()`, you can use an optional utterance identifier to be notified on the completion of the synthesis to the file:

```
HashMap<String, String> myHashRender = new HashMap();
String wakeUpText = "Are you up yet?";
String destFileName = "/sdcard/myAppCache/wakeUp.wav";
myHashRender.put(TextToSpeech.Engine.KEY_PARAM_UTTERANCE_ID, wakeUpText);
mTts.synthesizeToFile(wakuUpText, myHashRender, destFileName);
```

Once you are notified of the synthesis completion, you can play the output file just like any other audio resource with android.media.MediaPlayer.

But the `TextToSpeech` class offers other ways of associating audio resources with speech. So at this point we have a WAV file that contains the result of the synthesis of "Wake up" in the previously selected language. We can tell our TTS instance to associate the contents of the string "Wake up" with an audio resource, which can be accessed through its path, or through the package it's in, and its resource ID, using one of the two `addSpeech()` methods:

```
mTts.addSpeech(wakeUpText, destFileName);
```

This way any call to speak() for the same string content as `wakeUpText` will result in the playback of `destFileName`. If the file is missing, then speak will behave as if the audio file wasn't there, and will synthesize and play the given string. But you can also take advantage of that feature to provide an option to the user to customize how "Wake up" sounds, by recording their own version if they choose to. Regardless of where that audio file comes from, you can still use the same line in your Activity code to ask repeatedly "Are you up yet?":

```
mTts.speak(wakeUpText, TextToSpeech.QUEUE_ADD, myHashAlarm);
```

When not in use...

The text-to-speech functionality relies on a dedicated service shared across all applications that use that feature. When you are done using TTS, be a good citizen and tell it "you won't be needing its services anymore" by calling `mTts.shutdown()`, in your Activity `onDestroy()` method for instance.

Conclusion

Android now talks, and so can your apps. Remember that in order for synthesized speech to be intelligible, you need to match the language you select to that of the text to synthesize. Text-to-speech can help you push your app in new directions. Whether you use TTS to help users with disabilities, to enable the use of your application while looking away from the screen, or simply to make it cool, we hope you'll enjoy this new feature.

Android 1.6 archive How-to Text-to-Speech

Newer post **Older post**

Android Developers

Newsletter




Google Play