MusicPlayerApp

Step 1: Adding songs in App

* Create a new resource folder "raw" of type *raw.*
* Add music or songs in this folder(by copy paste).

Step 2: Creating layout to play music.

Image

<<

>>

**MediaPlayer**

One of the most important components of the media framework is the MediaPlayer class. An object of this class can fetch, decode, and play both audio and video with minimal setup. It supports several different media sources such as:

Local resources

Internal URIs, such as one you might obtain from a Content Resolver

External URLs (streaming)

For a list of media formats that Android supports, see the Android Supported Media Formats document.

Here is an example of how to play audio that's available as a local raw resource (saved in your application's res/raw/ directory):

MediaPlayer mediaPlayer = MediaPlayer.create(context, R.raw.sound\_file\_1);

mediaPlayer.start(); // no need to call prepare(); create() does that for you

In this case, a "raw" resource is a file that the system does not try to parse in any particular way. However, the content of this resource should not be raw audio. It should be a properly encoded and formatted media file in one of the supported formats.

And here is how you might play from a URI available locally in the system (that you obtained through a Content Resolver, for instance):

Uri myUri = ....; // initialize Uri here

MediaPlayer mediaPlayer = new MediaPlayer();

mediaPlayer.setAudioStreamType(AudioManager.STREAM\_MUSIC);

mediaPlayer.setDataSource(getApplicationContext(), myUri);

mediaPlayer.prepare();

mediaPlayer.start();

Playing from a remote URL via HTTP streaming looks like this:

String url = "http://........"; // your URL here

MediaPlayer mediaPlayer = new MediaPlayer();

mediaPlayer.setAudioStreamType(AudioManager.STREAM\_MUSIC);

mediaPlayer.setDataSource(url);

mediaPlayer.prepare(); // might take long! (for buffering, etc)

mediaPlayer.start();

Note: If you're passing a URL to stream an online media file, the file must be capable of progressive download.

Caution: You must either catch or pass IllegalArgumentException and IOException when using setDataSource(), because the file you are referencing might not exist.

**Asynchronous Preparation**

Using MediaPlayer can be straightforward in principle. However, it's important to keep in mind that a few more things are necessary to integrate it correctly with a typical Android application. For example, the call to prepare() can take a long time to execute, because it might involve fetching and decoding media data. So, as is the case with any method that may take long to execute, you should never call it from your application's UI thread. Doing that will cause the UI to hang until the method returns, which is a very bad user experience and can cause an ANR (Application Not Responding) error. Even if you expect your resource to load quickly, remember that anything that takes more than a tenth of a second to respond in the UI will cause a noticeable pause and will give the user the impression that your application is slow.

To avoid hanging your UI thread, spawn another thread to prepare the MediaPlayer and notify the main thread when done. However, while you could write the threading logic yourself, this pattern is so common when using MediaPlayer that the framework supplies a convenient way to accomplish this task by using the prepareAsync() method. This method starts preparing the media in the background and returns immediately. When the media is done preparing, the onPrepared() method of the MediaPlayer.OnPreparedListener, configured through setOnPreparedListener() is called.

**Managing State**

Another aspect of a MediaPlayer that you should keep in mind is that it's state-based. That is, the MediaPlayer has an internal state that you must always be aware of when writing your code, because certain operations are only valid when then player is in specific states. If you perform an operation while in the wrong state, the system may throw an exception or cause other undesireable behaviors.

The documentation in the MediaPlayer class shows a complete state diagram, that clarifies which methods move the MediaPlayer from one state to another. For example, when you create a new MediaPlayer, it is in the Idle state. At that point, you should initialize it by calling setDataSource(), bringing it to the Initialized state. After that, you have to prepare it using either the prepare() or prepareAsync() method. When the MediaPlayer is done preparing, it will then enter the Prepared state, which means you can call start() to make it play the media. At that point, as the diagram illustrates, you can move between the Started, Paused and PlaybackCompleted states by calling such methods as start(), pause(), and seekTo(), amongst others. When you call stop(), however, notice that you cannot call start() again until you prepare the MediaPlayer again.

Always keep the state diagram in mind when writing code that interacts with a MediaPlayer object, because calling its methods from the wrong state is a common cause of bugs.

**Releasing the MediaPlayer**

A MediaPlayer can consume valuable system resources. Therefore, you should always take extra precautions to make sure you are not hanging on to a MediaPlayer instance longer than necessary. When you are done with it, you should always call release() to make sure any system resources allocated to it are properly released. For example, if you are using a MediaPlayer and your activity receives a call to onStop(), you must release the MediaPlayer, because it makes little sense to hold on to it while your activity is not interacting with the user (unless you are playing media in the background, which is discussed in the next section). When your activity is resumed or restarted, of course, you need to create a new MediaPlayer and prepare it again before resuming playback.

Here's how you should release and then nullify your MediaPlayer:

mediaPlayer.release();

mediaPlayer = null;

As an example, consider the problems that could happen if you forgot to release the MediaPlayer when your activity is stopped, but create a new one when the activity starts again. As you may know, when the user changes the screen orientation (or changes the device configuration in another way), the system handles that by restarting the activity (by default), so you might quickly consume all of the system resources as the user rotates the device back and forth between portrait and landscape, because at each orientation change, you create a new MediaPlayer that you never release. (For more information about runtime restarts, see Handling Runtime Changes.)