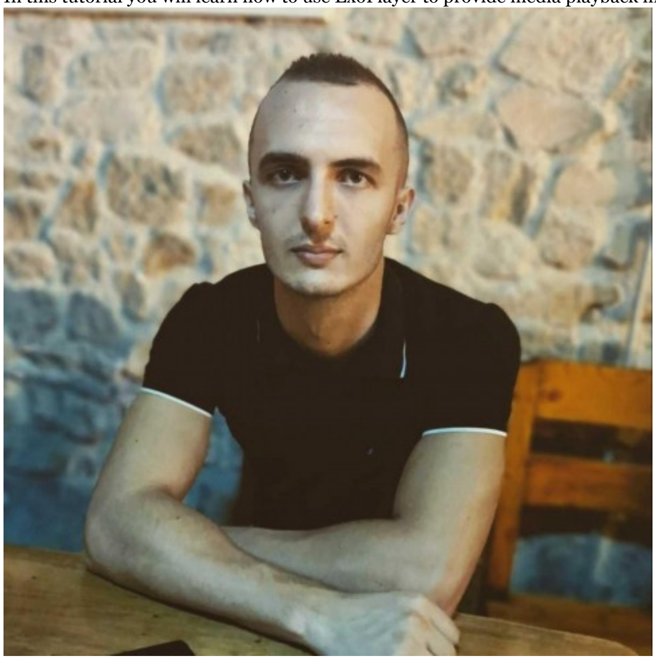
Home · Android & Kotlin Tutorials

Media Playback on Android with ExoPlayer: Getting Started

In this tutorial you will learn how to use ExoPlayer to provide media playback in your Android app.



By Dean Djermanović Nov 14 2018 · Article (25 mins) · Beginner

4.7/5

6

Ratings

Playing music or videos is a very popular activity on Android devices. There are several ways of providing media playback in your Android app and, in this tutorial, you'll learn how to use *ExoPlayer* to do this.

ExoPlayer is a media playback library for Android which provides an alternative to Android's MediaPlayer API. ExoPlayer is used by YouTube and Play Movies for video playback.

In this tutorial, you'll build an Android app that allows you to watch videos from *Cloudinary*'s free cloud-based video management solution. In the process, you'll learn:

- [illegible]
- [illegible]
- [illegible]
- [illegible]

Note: This tutorial assumes that you are familiar with the basics of Android development. If you are completely new to Android development, read through our Beginning Android Development tutorials first.

Media Playback on the Android Framework

Audio and Video

First, you'll do a little experiment. Take out your mobile device and count the number of apps you interact with on a regular basis. How many of them have the ability to play some kind of media? In fact, count how many of them don't have that possibility. You may find that the number is zero — it seems every single app a user interacts with on a daily basis can play some kind of media. This demonstrates just how important and popular media playback is on mobile devices.
But no matter the media type, they all have one thing in common: They have to get the media *from* somewhere to play it on your device.
Media used in an app can be stored on the local device's storage. That same app can also allow you to stream the media from the Internet; in that case, some remote web server is a media source. Media can also be available through many other streaming technologies, but that's beyond the scope of this tutorial.
So how do you play the media from any source? Before the hands-on part of this tutorial, you'll first cover the different ways media can be played on the Android framework.

Playing Media on Android

The Android framework provides several options for media playback, covered below.

Sending Implicit Intent

An `Intent` represents an app's "intent to do something." As with all implicit `Intents`, you have to specify a general action that you want to perform.
For media playback, you must specify the `ACTION_VIEW`. You also have to include a `URI` of the media sample that you want to play. If there's an app on the device that can handle that media type, the Android system launches it. This is useful for very simple use cases, the downside is you don't play the media within your app — you use another app on the device that was built to handle the `Intent`.

YouTube Player API

To play a YouTube video, you can send an implicit `Intent` to launch the YouTube app, which will then play a video. But you don't have to do that. The *YouTube Android Player API* provides an embedded player to play YouTube videos directly in your app, and it gives you the possibility to customize the playback experience. This is useful if your use case is to play videos specifically from YouTube.

MediaPlayer

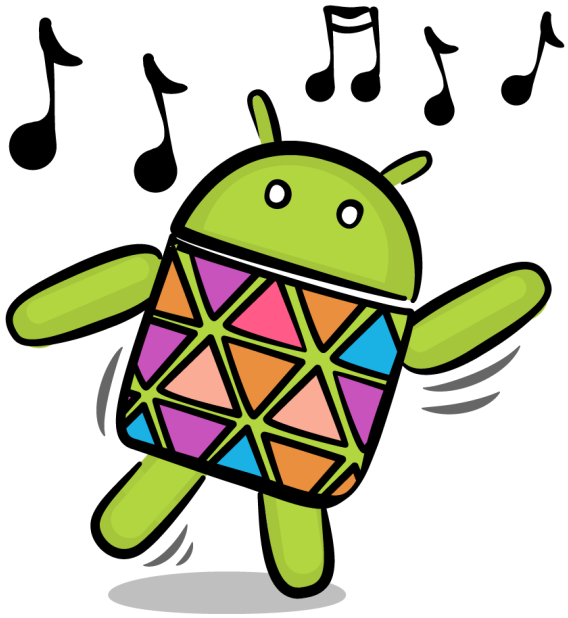
The Android multimedia framework includes support for playing a variety of common media types so that you can easily integrate audio, video and images into your apps. The *MediaPlayer* class is the primary entry point for playing sound and video. It supports the most common audio and video formats and data sources and because of that is good enough for many simple use cases. MediaPlayer is also very straightforward to use, but the downside is that it supports very little customization.

ExoPlayer

ExoPlayer is an open-source library that provides an alternative to Android's MediaPlayer API for playing audio and video. ExoPlayer supports features not supported by Android's MediaPlayer API, which you'll see later, and it's also easy to customize and extend.
Because of that, ExoPlayer is recommended for media player apps of any complexity on Android. ExoPlayer's standard audio and video components are built on Android's MediaCodec API, which was released in Android 4.1 (API level 16), which means ExoPlayer can only be used on devices running Android 4.1 or greater. This is also what you'll use in this tutorial.

Custom Player

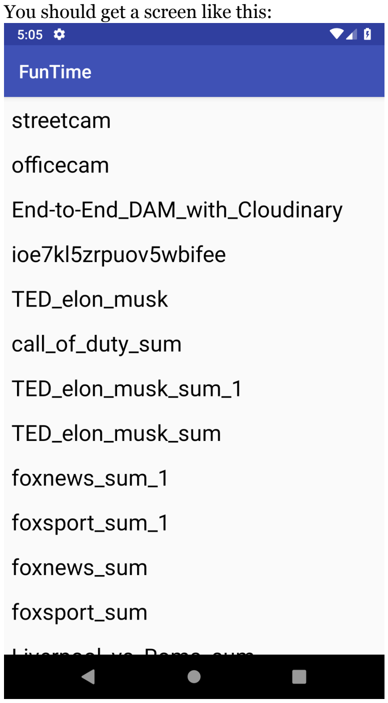
It is also possible to create a custom media player from low-level media APIs. The downside of this is that it's very complicated and, in most cases unnecessary, other media-playing possibilities are good enough for almost every use case. Don't reinvent the wheel. :]



Getting Started

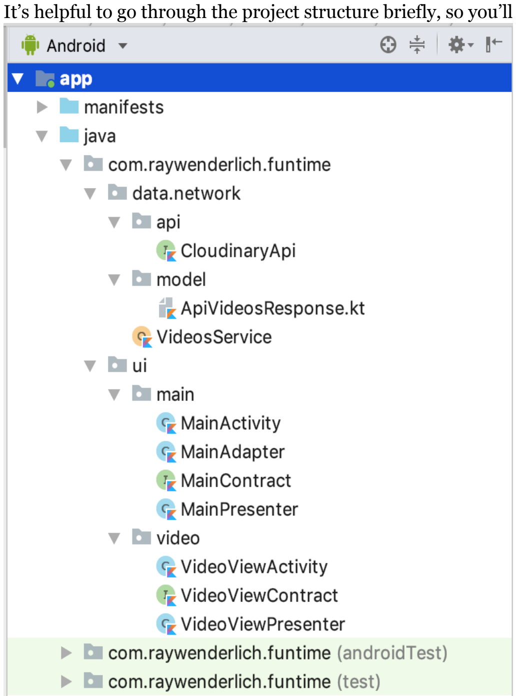
To show you how to implement ExoPlayer in an app, you'll create a simple app called *FunTime* that allows you to play videos from Cloudinary directly from the app.
You'll use sample videos from Cloudinary as your media source. You're not required to create an account for this library.
Download the materials for this tutorial using the *Download Materials* button at the top or bottom of the page. Open the starter project in Android Studio 3.0 or greater.
Now, build and run your app to make sure everything works as expected.

You should get a screen like this:



Project Structure

It’s helpful to go through the project structure briefly, so you’ll do that now.



The package names are pretty self-explanatory, but there are a few things that you need to notice, here.
The app is written using the *Model-View-Presenter (MVP) architectural pattern*. MVP is great way of organizing your code in Android. Some important advantages of using MVP are:

[illegible]

[illegible]

[illegible]

Adding a Media Player to the Application

The main screen shows you a list of sample video names that are fetched from Cloudinary.
When you click on a list item `VideoViewActivity` launches, but shows nothing in the starter project. This is where your video is going to be displayed.

Adding the Dependency

Recall that ExoPlayer is a library, in order to use it you have to add it to the project first. The ExoPlayer library is split into modules to allow developers to import only a subset of the functionality provided by the full library. The benefits of depending on only the modules you need are that you get a smaller APK size and you don't include the features in your app that you aren't going to use.
These are the available modules and their purpose:

`exoplayer-core` [illegible]
`exoplayer-dash` [illegible]
`exoplayer-hls` [illegible]
`exoplayer-smoothstreaming` [illegible]
`exoplayer-ui` [illegible]

It's still possible to depend on the full library if you prefer which is equivalent to depending on all of the modules individually.
For the sake of simplicity we'll add the full library.
Open your app module level `build.gradle` file and add the following dependency to the `dependencies` block:

```
implementation 'com.google.android.exoplayer:exoplayer:' + project.ext.exoPlayerVersion
```

The ExoPlayer version constant is already added to the project level `build.gradle` file so you can just use that version.
Sync the project after adding the dependency.

Creating the View

Next, you'll create the view. If you were using Android's MediaPlayer API you would display videos in a `SurfaceView`. The ExoPlayer library provides it's own high level view for media playback. It displays video, subtitles and album art, and also displays playback controls.
To add it, open the `activity_video_view.xml` layout file from `res/layout` and replace the contents with the following:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".ui.video.VideoViewActivity">

  <com.google.android.exoplayer2.ui.PlayerView
    android:id="@+id/ep_video_view"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

</android.support.constraint.ConstraintLayout>
```

Open the *VideoViewActivity.kt* file in the `iu.video` package and add a property for the `PlayerView`:

```
private lateinit var videoView: PlayerView
```

Initialize the view in the `init()` method:

```
videoView = findViewById(R.id.ep_video_view)
```

Creating the Player

Since you're using the MVP pattern in this project, you will decouple the view from the player. Start by creating a new `com.raywenderlich.funtime.device.player` package.
Inside this package, create a `MediaPlayer` interface, which is going to describe the behavior for the media player, and a `MediaPlayerImpl` class, which will contain the concrete implementation of your media player. Make the `MediaPlayerImpl` class implement the `MediaPlayer` interface.
Using the `MediaPlayer` interface makes swapping player implementations a breeze. You might want to explore creating alternate implementations without using ExoPlayer to explore the Android Media APIs more deeply.
Now, open the `MediaPlayerImpl` class. First, you need to initialize your player.
Add a property called `exoPlayer` for the player:

```
private lateinit var exoPlayer: ExoPlayer
```

Also add a property for the context that you'll set and use later:

```
private lateinit var context: Context
```

Next, add the `initializePlayer()` method where you're going to create a new instance of ExoPlayer and assign it to the exoPlayer member variable.
You can create an `ExoPlayer` instance using `ExoPlayerFactory`. The factory provides a range of methods for creating `ExoPlayer` instances with varying levels of customization. But, for most use cases, you should use one of the `ExoPlayerFactory.newSimpleInstance` methods.
Initialize `exoPlayer` in the method like this:

```
private fun initializePlayer() {

  val trackSelector = DefaultTrackSelector()
  val loadControl = DefaultLoadControl()
  val renderersFactory = DefaultRenderersFactory(context)

  exoPlayer = ExoPlayerFactory.newSimpleInstance(
      renderersFactory, trackSelector, loadControl)
```

```
}
```

`ExoPlayerFactory.newSimpleInstance()` takes three parameters:

- [illegible] `ExoPlayer` [illegible]
- [illegible]
- [illegible]

Don't worry about the specifics of these classes; using the default classes works perfectly in most use cases.
Awesome, you created an instance of the `ExoPlayer`!
You want your player to have the ability to play media, so describe that in the `MediaPlayer` interface by adding the following method:

```
fun play(url: String)
```

Implement that method in the `MediaPlayerImpl` class.
This is how you play the media with `ExoPlayer`:

```
override fun play(url: String) {
  //1
  val userAgent = Util.getUserAgent(context, context.getString(R.string.app_name))
  //2
  val mediaSource = ExtractorMediaSource
      .Factory(DefaultDataSourceFactory(context, userAgent))
      .setExtractorsFactory(DefaultExtractorsFactory())
      .createMediaSource(Uri.parse(url))
  //3
  exoPlayer.prepare(mediaSource)
  //4
  exoPlayer.playWhenReady = true
}
```

Going through this step by step:

1. `UserAgent` [illegible]
2. `ExoPlayer` [illegible] `MediaSource` [illegible] `MediaSource` [illegible] `ExtractorsFactory` [illegible] `createMediaSource()` [illegible] `Uri` [illegible]
3. [illegible] `prepare()` [illegible] `ExoPlayer` [illegible]
4. [illegible] `playWhenReady` [illegible] `true` [illegible] `false` [illegible]

You have now initialized the player and you have the view. What's next?

Attaching the Player to a View

Attaching the player to the view is very straightforward. You just set the `ExoPlayer` instance on the player view that you added to the xml by calling the `setPlayer(...)` method.
Since you're using MVP and you're decoupling the concrete player implementation from the view, you need a way to get the underlying player implementation.
Add a method to the `MediaPlayer` interface that will give you access to the underlying implementation:

```
fun getPlayerImpl(context: Context): ExoPlayer
```

Implement that method in `MediaPlayerImpl` class:

```
override fun getPlayerImpl(context: Context): ExoPlayer {
  this.context = context
  initializePlayer()
  return exoPlayer
}
```

Now, you have access to the `ExoPlayer` implementation. `VideoViewActivity` will get the `ExoPlayer` instance through `VideoViewPresenter`.
Add a `getPlayer()` method to the `VideoViewContract.Presenter` interface in the `ui.video` package, which returns a `MediaPlayer` instance:

```
fun getPlayer(): MediaPlayer
```

Add a media player property to the `VideoViewPresenter`:

```
private val mediaPlayer = MediaPlayerImpl()
```

Implement the `getPlayer()` method, which will just return the media player instance:

```
override fun getPlayer() = mediaPlayer
```

In `VideoViewActivity`, set the player on the view inside the `init()` method by calling:

```
videoView.player = presenter.getPlayer().getPlayerImpl(this)
```

To actually play the video, add a `play()` method to the `VideoViewContract.Presenter` interface and pass in the media url:

```
fun play(url: String)
```

Now, implement that method in the `VideoViewPresenter`. This method just delegates media playing to media player.

```
override fun play(url: String) = mediaPlayer.play(url)
```

Great, now you're ready to play the video.
At the end of `VideoViewActivity`'s `init()` method, tell the presenter to play the video:

```
presenter.play(videoUrl)
```

It's important to release the player when it's no longer needed, in order to free up limited resources, such as video decoders, for use by other apps. This can be done by calling `ExoPlayer.release()`.
Add a `releasePlayer()` method to the `MediaPlayer` interface:

```
fun releasePlayer()
```

And implement it in the `MediaPlayerImpl` class:

```
override fun releasePlayer() {
  exoPlayer.stop()
  exoPlayer.release()
}
```

Add the `releasePlayer()` method to the `VideoViewContract.Presenter`, as well, and implement it in the `VideoViewPresenter` class:

```
override fun releasePlayer() = mediaPlayer.releasePlayer()
```

You need to make sure that `VideoViewActivity` releases the player when it is no longer the active Activity.
To do this, you release the player in `onPause()` if on Android Marshmallow and below:

```
override fun onPause() {
  super.onPause()
  if (Build.VERSION.SDK_INT < Build.VERSION_CODES.N) {
    presenter.releasePlayer()
 }
}
```

Or release in `onStop` if on Android Nougat and above because of the multi window support that was added in Android N:

```
override fun onStop() {
  super.onStop()
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.N) {
    presenter.releasePlayer()
 }
}
```

Build and run your app to see what happens.
Click on any item in the list and you will get a screen like this:

Now you can play video. Awesome. :]
Guess we're done here, right? Not yet.

Customizing ExoPlayer

If you take a look at the UI of `ExoPlayer`, it's minimalistic. Next, you'll make it nice and shiny.
The `ExoPlayer` library is designed specifically with customization in mind. That's a huge advantage of `ExoPlayer`: you can customize almost anything. The `ExoPlayer` library defines a number of interfaces and abstract base classes that make it possible for app developers to easily replace the default implementations provided by the library.
In your app, you'll customize the user interface.

Changing the Appearance

Video is displayed in the `PlayerView` in XML. `PlayerView` is a high level UI component for media playback which displays the video and playback controls. Playback controls are displayed in a `PlaybackControlView`. Those elements support a variety of XML attributes, which you can use to customize the look of the UI.
You can also override the default layout files. When these views are inflated, they use specific layout files that determine how the UI will look. You'll change the appearance of the playback controls.
When `PlaybackControlView` is inflated, it uses `exo_playback_control_view.xml`. Create a new XML layout file in the `res/layout` folder and name it `exo_playback_control_view.xml`. This will override the default file.
Update the file to the following:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:layout_gravity="bottom"
 android:background="#CC000000"
 android:layoutDirection="ltr"
 android:orientation="vertical">

 <LinearLayout
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:gravity="center"
   android:orientation="horizontal"
   android:paddingTop="4dp">

   <ImageButton
     android:id="@id/exo_rew"
     style="@style/ExoMediaButton.Rewind" />

   <ImageButton
     android:id="@id/exo_play"
     style="@style/CustomExoMediaButton.Play" />

   <ImageButton
     android:id="@id/exo_pause"
     style="@style/CustomExoMediaButton.Pause" />

   <ImageButton
     android:id="@id/exo_ffwd"
     style="@style/ExoMediaButton.FastForward" />
 </LinearLayout>

 <LinearLayout
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:layout_marginTop="4dp"
   android:gravity="center_vertical"
   android:orientation="horizontal">

   <TextView
     android:id="@id/exo_position"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:includeFontPadding="false"
     android:paddingLeft="4dp"
     android:paddingRight="4dp"
     android:textColor="#FFBEBEBE"
     android:textSize="14sp"
     android:textStyle="bold" />

   <com.google.android.exoplayer2.ui.DefaultTimeBar
     android:id="@id/exo_progress"
     android:layout_width="0dp"
     android:layout_height="26dp"
     android:layout_weight="1" />
```

```
    <TextView
      android:id="@id/exo_duration"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:includeFontPadding="false"
      android:paddingLeft="4dp"
      android:paddingRight="4dp"
      android:textColor="#FFBEBEBE"
      android:textSize="14sp"
      android:textStyle="bold" />

  </LinearLayout>
</LinearLayout>
```

These changes customize how the Play and Pause buttons look. Open `styles.xml` and view the custom styles for those buttons. The default styles change the drawable source that is displayed and make the buttons a little bit bigger.

```
<style name="CustomExoMediaButton">
    <item name="android:background">?android:attr/selectableItemBackground</item>
    <item name="android:scaleType">fitXY</item>
    <item name="android:layout_width">@dimen/video_view_exo_player_play_pause_button_dimen</item>
    <item name="android:layout_height">@dimen/video_view_exo_player_play_pause_button_dimen</item>
</style>

<style name="CustomExoMediaButton.Play">
    <item name="android:src">@drawable/ic_play_circle_filled</item>
</style>

<style name="CustomExoMediaButton.Pause">
    <item name="android:src">@drawable/ic_pause_circle_filled</item>
</style>
```

Build and run your app and play a video to see what it looks like.

OK, it's not as nice and shiny as you might have hoped, but feel free to modify the styles if you want to see a more dramatic change. :]
There is a small issue with this approach. Since you overrode the default XML layout file, all instances of the `PlaybackControlView` in your app will be customized like this. If you don't want this behavior, you can customize individual instances as well. You can use the `controller_layout_id` attribute in the XML to specify a custom layout file for individual instances.

Pros and Cons of ExoPlayer

The biggest advantages of `ExoPlayer` are its flexibility and rich feature stack, but that also makes it harder to work with it.
Since you can customize the player to suit almost every use case, `ExoPlayer` is the best choice for complex use cases. For simple use cases there really isn't a reason to use `ExoPlayer`, `MediaPlayer` will suffice.
For audio-only playback on some devices, `ExoPlayer` may consume significantly more battery than `MediaPlayer`.
One more advantage of `MediaPlayer` over `ExoPlayer` is that `MediaPlayer` works all the way back to the beginning of Android, while `ExoPlayer` is only available on Jelly Bean and above. But I wouldn't call this a problem since there's only about 1% of active devices running earlier versions.

Where to Go From Here?

You covered a lot in this tutorial, but `ExoPlayer` has many other possibilities and advanced features that aren't mentioned here. In general, if there is something that you can't do with Android's `MediaPlayer` there's a high probability that you can do it with `ExoPlayer`. Therefore, `ExoPlayer` is often the best choice for media playback on Android.

However, be careful of over-engineering! You must be thinking now: "ExoPlayer is awesome, I'll use it all the time!" Before you do that, ask yourself this: "Do I really need an ExoPlayer?" Say you have a killer app idea. You want to make an app that plays silly sound effects. Do you really need `ExoPlayer` here? `ExoPlayer` has many cool features, but in this case you don't need it. You just need a way to play a very simple sound. Android's `MediaPlayer` would be a better choice. Don't over-engineer things!

The FunTime app was just one example of how you can play videos from a remote web server in your app. If you want to check out other features of `ExoPlayer` and see how to implement those, Google's codelab is a good place to start. You can check it out here.

We hope you enjoyed this tutorial and learned something from it. If you have any question or comments, or you want to share your experience with `ExoPlayer`, please join in the discussion below.