Introduction to Android

Implementing a Video Player

Mobile Programming
2^e année SN, Parcours Multimédia - Informatique

Sessions 1-3

Version: v2024.1.0-rc1
(master @ 98c3c4c 2024-03-02)

Contents

Objective

The objective of these first three TP sessions is to get you familiar with the Android development environment and to implement a multimedia application. In particular we will develop step by step an application to play videos from the internal SD card of the device as well as streaming remote videos from the Internet.

References

Here are some useful resources that may help you to complete the TP:

- A reminder of the main components of an Android application
  `https://developer.android.com/guide/components/fundamentals.html`
- A quick introduction to Android Studio
  `http://developer.android.com/tools/studio/index.html`
- Some explained examples of applications with specific components
  `https://developer.android.com/samples/index.html`
- The official documentation of the Android API.
  `https://developer.android.com/reference/packages.html`

1 First Application: web browsing using Intents

In this first warm-up exercise, you are going to develop a simple application to visit web pages: the user types the URL of a web page and a button launches the Android default Web Browser to load the page.

1.1 Getting started with Android Studio

Before starting please clean up your account and free as much space as possible (empty the trash can, remove all teaching unrelated files *etc.*).

Android Studio is the official IDE for Android application development, based on IntelliJ IDEA, a Java IDE. Android Studio offers the same functionalities as IntelliJ for editing Java applications along with many other tools that have been added to support the development, debugging and testing of Android applications. In particular, it integrates Gradle as a building system, a rich layout editor to create the application GUI, and tools for logging the applications. We will discover all these tools as we create our first application.

In order to set up Android Studio, **only the very first time you use it**, you need to run the following script:

```
echo "export ANDROID_AVD_HOME=/work/avd" >> ~/.bashrc && mkdir -p /work/avd
/mnt/n7fs/local/bin/initAndroidStudio && source ~/.bashrc
```

After that, you can launch Android Studio with:

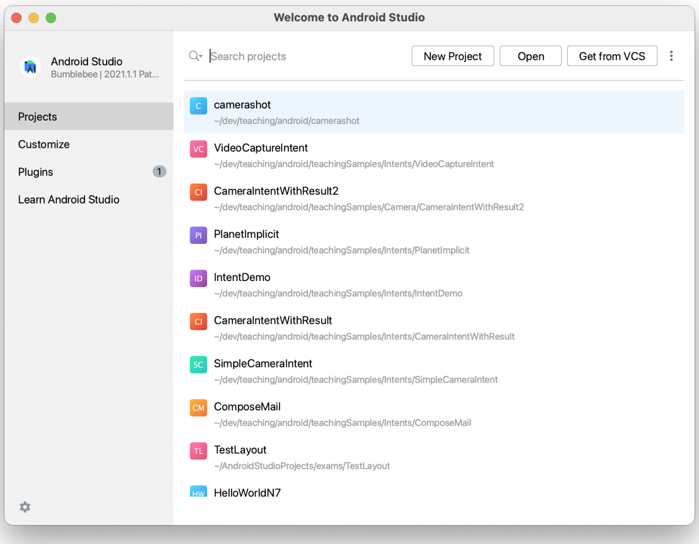
```
studio
```

Normally, a wizard similar to the one in Figure 1 should pop up. The wizard allows to create a new project, load an existing one, and other things that for the moment are out of the scope of this lab session.

1.2 Create a new Android Studio project

Thanks to Android Studio, we can create a new Android application stub in quite few steps. From the “Quick Start” panel of the wizard click “Start a new Android Studio project” to start the wizard that creates all the files needed for the application. Following the various steps that are proposed:

- create a new project with an **“Empty Activity“** with the following settings:
  - Application name: `Web Browser`.
  - Package name: `YOUR_LOGIN.enseeiht.fr`
  - Language: `Java` **NOT KOTLIN!**
  - Minimum SDK: `API 16: Android 4.1 (Jelly Bean)`



camerashot
~/dev/teaching/android/camerashot

VideoCaptureIntent
~/dev/teaching/android/teachingSamples/Intents/VideoCaptureIntent

CameraIntentWithResult2
~/dev/teaching/android/teachingSamples/Camera/CameraIntentWithResult2

PlanetImplicit
~/dev/teaching/android/teachingSamples/Intents/PlanetImplicit

IntentDemo
~/dev/teaching/android/teachingSamples/Intents/IntentDemo

CameraIntentWithResult
~/dev/teaching/android/teachingSamples/Intents/CameraIntentWithResult

SimpleCameraIntent
~/dev/teaching/android/teachingSamples/Intents/SimpleCameraIntent

ComposeMail
~/dev/teaching/android/teachingSamples/Intents/ComposeMail

TestLayout
~/AndroidStudioProjects/exams/TestLayout

HelloWorldN7

Figure 1: The wizard that appears when you launch Android Studio for the first time.

Once you complete the wizard, Android Studio will generate all the classes and all the resources needed for the application, and it will show them in the editor. Let's have a quick look at the Android Studio editor.

1.3 Android Studio editor

Take a moment to look at the Android Studio editor. As it has been explained in class, the interface of the editor is composed of 3 main parts, the project views, the editor itself, and the Android toolbar for logging and debugging. By default, Android Studio displays your project files in the *Android* project view (see Figure 2). This view shows a flattened version of your project's structure that provides quick access to the key source files of Android projects and helps you work with the Gradle-based build system. The Android project view:

- Shows the most important source directories at the top level of the application hierarchy.
- Groups resource files for different locales, orientations, and screen types in a single group per resource type (see Figure 2). In particular, you can see that the application files are organized into 3 main folders:
  - `java/` contains the Java source files for the application.
  - `manifests/` contains the manifest files for the application.
  - `res/` contains the resource files for the application.
- Groups the build files for all modules in a common folder.

Another interesting view is the *Problem* view, which displays links to the source files containing any recognized coding and syntax errors, such as missing an XML element closing tag in a layout file.

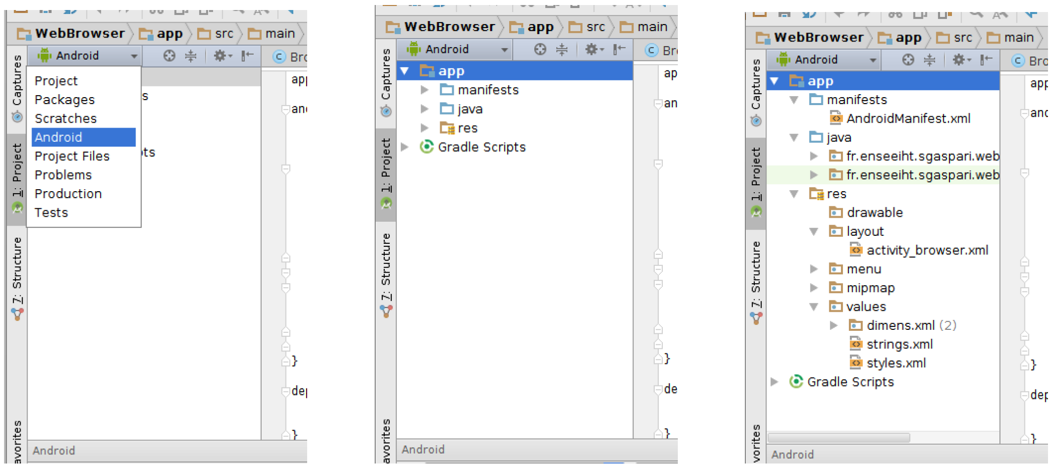


Figure 2: The “Android” project view displaying how the application is organized in different folders.

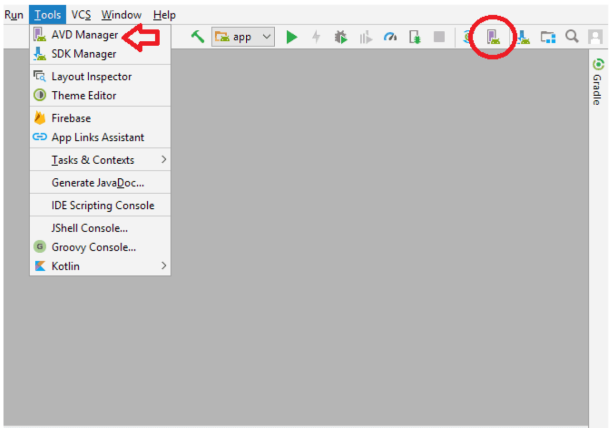


Figure 3: How to locate AVD Manager

1.4 The Virtual Device

We will use a virtual device to test our application. As we have seen in classes, Android Studio allows to create an emulator of an Android device quite easily:

- Open the AVD manager from the icon on the menu bar or from the menu Tools > Android > AVD Manager shown in Figure 3.
- Create a new device with a Nexus 5 device, with Nougat API 25 for x86_64 processor. It is located in the **x86 devices** tab and should be in bold.
- Once you created the emulator, you can start it by clicking on the green arrow (like the common “play” icon) in the AVD manager window.
- For your reference, here you can find the keyboard shortcuts to control the virtual device.

1.5 Start the application

The following text in red is only for N7 machines.
Due to some compatibility issues on the N7 machines, you need to look at the second `build.gradle` file in Gradle Scripts section in your file explorer and replace the dependencies section with a different version as shown here below:

```
dependencies {

    implementation 'androidx.appcompat:appcompat:1.2.0'
    implementation 'com.google.android.material:material:1.2.1'
    implementation 'androidx.constraintlayout:constraintlayout:2.0.1'
    testImplementation 'junit:junit:4.+'
    androidTestImplementation 'androidx.test.ext:junit:1.1.2'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.3.0'
}
```

Furthermore you also need to fix the build tools version by clicking on "File > Project Structure > Modules" and set `Build Tools Version` to 29.0.2 **instead of** 32.0.0**.**

In the AndroidManifest.xml located in the manifests folders, add `android:exported="true"` in the activity.
It should look similar to this:

```
<activity android:name="com.example.myActivity"
    android:exported="true">
```

Now you can finally run the application by clicking the Run icon (the green arrow) on the toolbox bar or from the Run > Run 'app' menu. Obviously, the application is pretty dumb and it does nothing. But now we have a stub from which we can develop our application.

1.6 The Log class and the LogCat view

Since Android applications run on external devices without a shell, the debugging could be troublesome. Fortunately, Android provides a logging system for collecting and viewing system debug output. Logs from various applications and portions of the system are collected in a series of buffers, which then can be viewed and filtered out using the LogCat view (you may need to click on the tab `6 Android` at the bottom of the window). Now let's try to see how the logging works:

- Android provides the class `android.util.Log` to send messages to the system. Check out the brief explanation here
  `http://developer.android.com/tools/debugging/debugging-log.html`
- define a private constant string named `TAG` containing an identifier for all the messages sent by our activity:

  ```
  private static final String TAG = "Browser Activity"
  ```

- now you can add a log message to the method `onCreate()`:

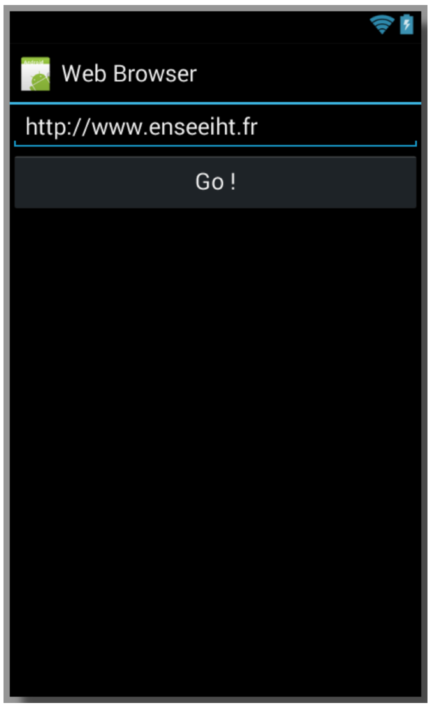


Figure 4: The graphical layout of the Web Browser.

```
Log.d( TAG, "OnCreate() called!" )
```

- Re-run the application and check the LogCat to see the message sent by the activity.
- Note that in this document the clickable hyperlink to the class API documentation is given for some classes and some class methods (*e.g.* `findViewById()`).

LogCat is a very useful tool to debug your application. Adding text messages may help you to follow the application flow and find the bug(s).

1.7 Setting up the application

Now let's start adding the elements to develop our browser application:

- open the layout from `res/layout` and switch to the Design Mode. If you get an error "Rendering Problems", be sure that the settings for the Android version are correct by selecting "API 22: Android 5.1.1" from the toolbar menu (see Figure 5)
- modify the GUI layout to add an editable field where the user can write the web address to load (`EditText`), and a button that asks the Web Browser to load the webpage (`Button`).
- the button must display the text "GO!", while the `EditText` must display the URL `http://www.enseeiht.fr` as default text so that it is displayed automatically when the application is launched for the first time; as we saw in classes, all the strings must be collected in `value/string.xml`[1].

[1]In order to create a new string you can either define it in the `strings.xml` and reference it in the `Text` attribute of the widget, or use the graphical GUI editor and follow the guided wizard by clicking on the small button in the `Text` attribute of the widget.

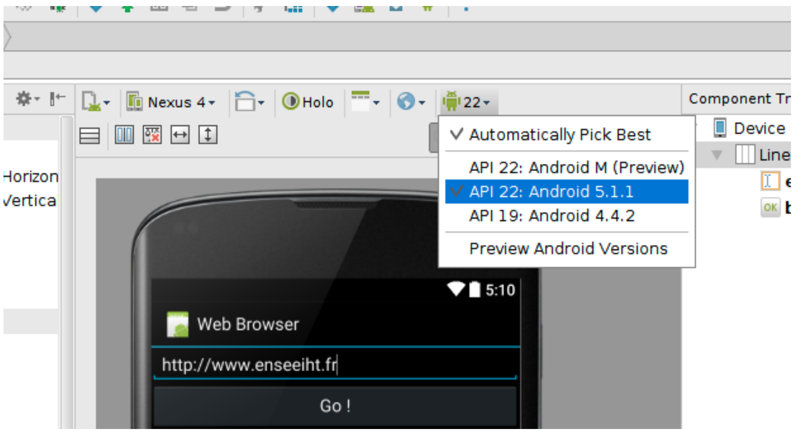


Figure 5: Check that the correct version of the API is set for the visualization in Design mode.

- Re-run the application and verify that the GUI is displayed correctly (see Figure 4).

1.8 Action and. . . GO!

The last thing to do is to wire the button so that the web browser is called when the user presses the button: to this end, we need to implement the button callback, and create the `Intent` that invokes the default Android browser.

- in `onCreate()` get the button from the resources using the function `findViewById()`;
- get also the reference to the text field, as we need it to get the web address typed by the user;
- once we have the reference to the button, set its callback listener using its method `setOnClickListener()`: among other choices, you can pass an anonymous class with `new View.OnClickListener()` and if you use the Android Studio auto-completion, it will automatically generate the stub of the method to implement (`OnClick()`);
- the callback listener has to create a new `Intent` to open the web address contained inside the `EditText`.

  An Intent is a messaging object you can use to request an action to be executed by another application component and it contains an abstract description of the operation to be performed. In this case you will need to use an implicit intent, as you are asking a general action to perform, which allows a component from any other available application to handle it.

  There are two main things that you need to specify when you create an intent, the action and the data.

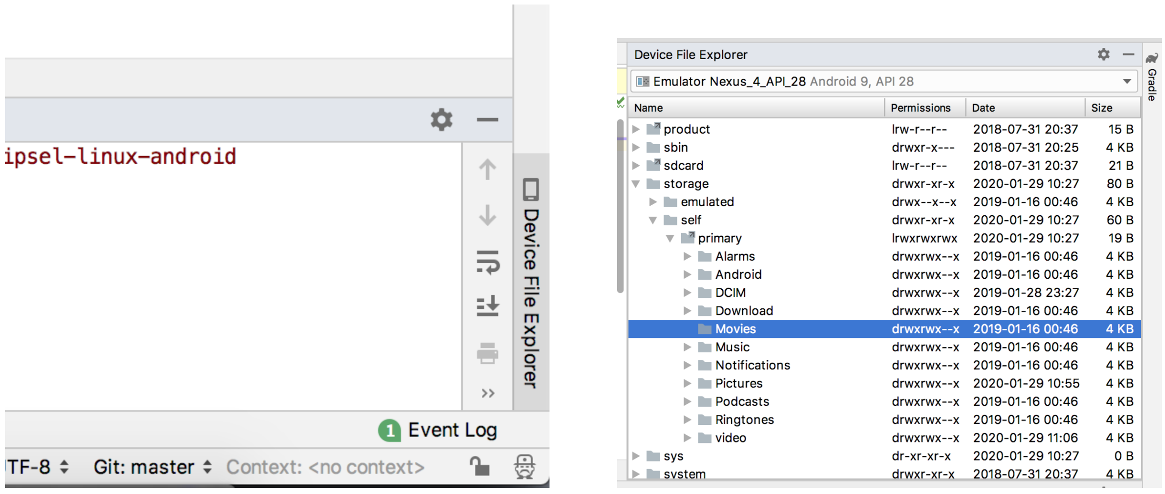
  – The **Action** that you are required to perform: when you create a new object of the class `Intent` you need to pass one of the possible predefined actions available in Android. For example, `ACTION_VIEW` is used when you have some information that activity can show to the user, such as a photo to view in a gallery application or an address to view in a map application; `ACTION_SEND` is used when you have some data that the user can share through another application, such as an email or social sharing application (see more here).

  - The **Data** is the URI (a `Uri` object) that references the data to be acted on and/or the MIME type of that data. The type of data supplied generally depends on the intent's action. For example, if the action is `ACTION_SEND`, the data should contain the URI of the document to edit. To set the data of the intent you can use either the intent method setData() or setDataAndType().

  Finally, once the intent is created and its data set, you can start the request with the method `startActivity()` of the class `Activity`

  Set up the intent action (`ACTION_VIEW`) and the data field as an URI obtained from `EditText` (remember to use the `Uri.parse()`). A quick review about `Intents` is available here
  `http://developer.android.com/guide/components/intents-filters.html`

- Re-run the application (. . . and debug it if necessary!)



Device File Explorer

Emulator Nexus_4_API_28 Android 9, API 28

Name	Permissions	Date	Size
product	lrw-r--r--	2018-07-31 20:37	15 B
sbin	drwxr-x---	2018-07-31 20:25	4 KB
sdcard	lrw-r--r--	2018-07-31 20:37	21 B
storage	drwxr-xr-x	2020-01-29 10:27	80 B
emulated	drwx--x--x	2019-01-16 00:46	4 KB
self	drwxr-xr-x	2020-01-29 10:27	60 B
primary	lrwxrwxrwx	2020-01-29 10:27	19 B
Alarms	drwxrwx--x	2019-01-16 00:46	4 KB
Android	drwxrwx--x	2019-01-16 00:46	4 KB
DCIM	drwxrwx--x	2019-01-28 23:27	4 KB
Download	drwxrwx--x	2019-01-16 00:46	4 KB
Movies	drwxrwx--x	2019-01-16 00:46	4 KB
Music	drwxrwx--x	2019-01-16 00:46	4 KB
Notifications	drwxrwx--x	2019-01-16 00:46	4 KB
Pictures	drwxrwx--x	2020-01-29 10:55	4 KB
Podcasts	drwxrwx--x	2019-01-16 00:46	4 KB
Ringtones	drwxrwx--x	2019-01-16 00:46	4 KB
video	drwxrwx--x	2020-01-29 11:06	4 KB
sys	dr-xr-xr-x	2020-01-29 10:27	0 B
system	drwxr-xr-x	2018-07-31 20:37	4 KB

Figure 6: To upload the video on the SSD, click "Device File Explore" on the bottom right side of Android Studio and then select the folder "storage > self > primary > Movies". You can either drag and drop the file on the folder or right-click on the folder, click on "Upload..." and select the file to upload.

2 Second Application: Building your custom Video Player

In this exercise, we want to create an application that plays a video using a custom video player. We will build the application incrementally:

- At first we develop a basic application that uses intent to select a video from the gallery and play it using the default Android Video Player;
- then we add a custom video player to our application.

2.1 Adding a video to the AVD SD card

Before starting, let's add a video on the virtual device SD card that we can use it to test our application:

- open the Device File Explore view from the bottom right side of Android Studio (see Figure 6 on the left);
- This view enables the user to see the content of the filesystem;
- select the folder "storage > self > primary > Movies" and add the video either by drag and drop or right-clicking on the folder and selecting "Upload...". The video can be found in `/mnt/n7fs/ens/tp_android/TP_01/documentariesandyou.mp4`
- you can check if the video has been correctly loaded by going on the device, `Settings` > `Storage` > `Internal Storage` > `Other` and you should see the video. If it not the case, you may need to migrate the data (Inside `Virtual SD Card` click on the menu on the top and select `Migrate Data`).
- if you want, you can add other video files (mind your disk quota...); here are the media formats supported by Android `http://developer.android.com/guide/appendix/media-formats.html`

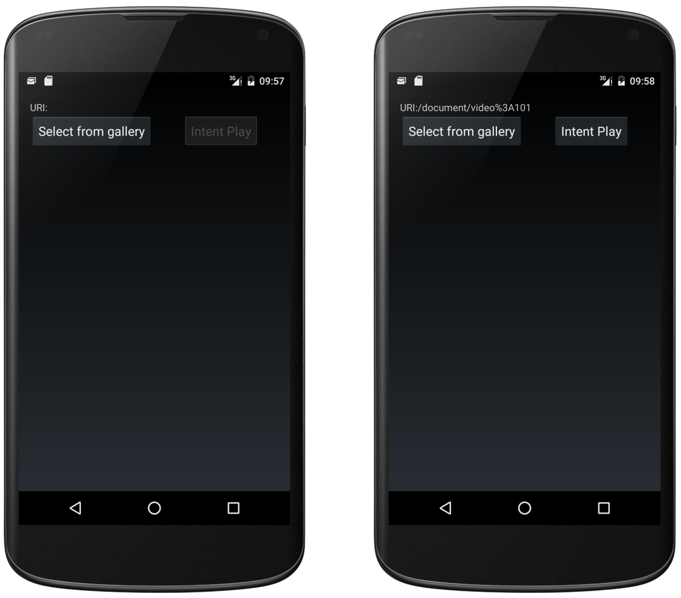


Figure 7: The initial GUI for the video player.

2.2 Playing videos using Intents

Similarly to the previous exercise, create a new Android project named `Video Player`:

- package name `fr.enseeiht.YOUR_LOGIN.player`,
- minimum SDK API 16,

For this first version of the application, the user must be able to pick a video from the application "Gallery" and play it with the default application "Video Player". We will use then an intent with the result to pick (the URI of) a video from the gallery and another intent to launch an application able to play it.

So let's start with the GUI. Figure 7 shows the GUI to implement, which contains:

- a `Button` Select from Gallery that allows the user to select a video from the gallery;
- a `Button` Intent play that is enabled only if the user has selected a video and when clicked will start an intent to play the selected video;
- two `TextView` used mostly for debugging purposes showing the URI of the selected video: the leftmost one just shows the label "URI:", the rightmost one will show the content of the URI when the video is selected.

2.2.1 Picking the video from gallery

We need to create an implicit `Intent` to ask for an available activity able to pick a video from all the videos available on the device and load the result on our video player.

- implement a new listener for the button Select from Gallery, and create a new implicit `Intent` to pick a video from the gallery:

  - the action related to the intent has to be the "pick a video" kind: look at the documentation of the action `ACTION_PICK`.
  - we need to use this constructor for the intent which specifies the URI where to pick from `Intent(String, Uri)`;
  - as URI we need to provide that of the directory of the media store containing the videos: `android.provider.MediaStore.Video.Media.EXTERNAL_CONTENT_URI`;
  - set the type of data to *any kind of video*, *i.e.* `"video/*"`: the activity that receives the intent will only display videos among which the user can choose.
  - here is a general reference about starting Activities to get a result http://developer.android.com/training/basics/intents/result.html
  - If you want to test the button with the intent pick, Android will propose to use either the "Video Player" or the "Photo" application to select the video. Use "Photo".

- start the activity with `startActivityForResult`, use an arbitrary value for the `requestCode`, *e.g.* you can declare a member constant for it

```
private static final int SELECT_VIDEO = 100;
```

- override the method `onActivityResult()` of the activity `PlayerActivity` to grab the result when the other activity is done. Verify that the request code matches with the one sent and that the user has actually selected an element (`resultCode`);
- the result intent gives back the URI of the selected video: store the result data from the intent in a class member and show its content inside the `TextView`;
- if the URI has been correctly retrieved enable the Intent play;
- run the application and test it.

2.2.2 Play the video

To play the video at the given URI when Intent play, we just need to set a callback for the button and start an intent as we did it for the first exercise. The intent contains as data the URI stored from the result of the "Pick" intent. **(Android will propose to use either the "Video Player" or the "Photo" application to play the video. Due to some issue related to permissions, only "Photo" will be able to play the video.)**

2.3 The custom video player

Now we can add the video playback feature directly in the main activity of our application. We first need to modify the layout to add the controls typical of a video player (see Figure 8 as reference):

- a new button Play from start;
- a new button Play / Resume;
- a `SurfaceView`, used to display the video;

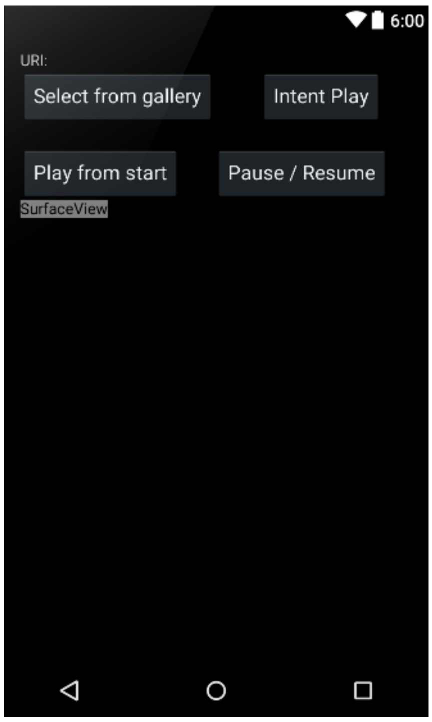


Figure 8: The graphical layout of the Video Player.

- Note that the main Activity has now to implement the class `SurfaceHolder.Callback`, *i.e.*

```
public class PlayerActivity extends Activity
                              implements SurfaceHolder.Callback {
...
```

2.3.1 The MediaPlayer class

Android provides the class `MediaPlayer` as the primary API for playing sound and video files. An object of this class can fetch, decode, and play both audio and video with minimal setup. Let's start setting up the graphical part to display the video.

2.3.1.1 Setting up the SurfaceView and the MediaPlayer

The `MediaPlayer` class continuously draws the video frames inside the `SurfaceView`. The `SurfaceView` provides a dedicated drawing surface embedded inside of the view hierarchy. In order to manage and access to the underlying surface, the `SurfaceHolder` interface is provided, which can be retrieved by calling `getHolder()`. Take a moment to read the "Class Overview" on the online documentation to better understand the use of these classes.

The surface holder must be initialized inside the `onCreate()` method of `PlayerActivity`:

```
// recover the SurfaceView from resources
surfaceView = (SurfaceView) findViewById( R.id.surfaceview );
// Get the surfaceHolder from it
surfaceHolder = surfaceView.getHolder();
// and assign to it the call back this class implements
surfaceHolder.addCallback( this );
// this is a compatibility check, setType has been deprecated since HoneyComb,
// it guarantees back compatibility
if( Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB )
     surfaceHolder.setType( SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS );
```

then the `SurfaceHolder` can be associated to the `MediaPlayer`

```
// the surfaceHolder.addCallback implemented by PlayerActivity
public void surfaceCreated( SurfaceHolder holder )
{
     mediaPlayer.setDisplay(surfaceHolder);
}
```

2.3.2 Playing the video

The `MediaPlayer` is a state machine that controls the audio/video playback. Again, take your time to read the online documentation of the class, in particular the state diagram (it may seems complicated at first, but it's straightforward and intuitive to follow).

- Add a callback listener for the Play from start button and implement the function that sets up the `MediaPlayer` and starts playing the video:
  - set URI as the source to be play by `MediaPlayer` (`setDataSource()`);
  - prepare the `MediaPlayer` for the playback (`prepare()`)
  - start the video (`start()`)
- Add another callback listener for the Pause button to pause (`pause()`) and resume (`start()`) the video according to its state; use `isPlaying()` to keep track of the player state.

Now you can try to run the application, select a video and play it.

2.3.2.1 The activity life-cycle To avoid any resource conflict, we need to stop and release the `MediaPlayer` whenever the user leaves the activity and saves the current position of the video playback, so that, when she comes back, the video resumes from the current position[2]. Therefore, we need to override the life-cycle callbacks `onPause()` and `onResume()` of `PlayerActivity` to manage the release and restart of `MediaPlayer`[3]:

[2] Contrary to audio playback, it makes less sense to keep the video playing in the background when the user leaves the application. As we saw in classes, to get the video to continue playing in the background we would rather need to implement a `Service`.

[3] Astuce: select the `PlayerActivity` class on the editor, press Ctrl + O : Android Studio will propose a wizard to automatically generate the code skeleton of the class method to override.

- whenever the user leaves the activity we need to save the current position (`getCurrentPosition()`) and release the `MediaPlayer` with `reset()` and `release()` . You can use a simple `int` variable to save the position.
- whenever the user comes back to the activity, we need to reinitialize the `MediaPlayer` (as we already did for the button listener) and start the video from the last saved position (use `seekTo()`).
- Note that for the Play from start to work when the video is playing or paused, or a new video is selected, a call to `reset()` is necessary before setting the data source.

2.3.3 The seek bar

We can use the `SeekBar` to navigate and jump to (or, "seek to") a specific position of the video. The `SeekBar` can be wired to the playback so that it displays the playback progress. We first implement the "seek to" function of the bar, and then we will create a thread dedicated to the position update.

SeekTo functionality Let's start by adding the SeekBar to the UI and wiring it so that we can skip the playback to any part of the video.

- Add a `SeekBar` to the interface: place it below the Play button and above the SurfaceView, so that it is centered and it is stretched on the whole width of the screen, *i.e.* `layout_alignParentStart="true"` and `layout_alignParentEnd="true"` (*c.f.* Figure 9).
- Inside `onCreate()` get the reference to the `SeekBar` from the resources, and set a new callback with `setOnSeekBarChangeListener()`: this manages the events related to the bar due to the user interaction.
- creates a new object implementing the interface class `OnSeekBarChangeListener` private to `PlayerActivity`; Android Studio creates the stubs for the 3 methods to implement. We are interested only in the method `onProgressChange` : the method is invoked whenever the progress level changes, and can distinguish between user-initiated changes (*i.e.* the user touched the bar) and those occurred programmatically (*i.e.* the playback position has been updated).
- if the change occurred because of the user interaction, set the new playback position in `MediaPlayer` (`seekTo(progress)`);

SeekBar Update To continuously update the playback position on the bar, we can create a thread to query the current position from `MediaPlayer` with a given frequency and update the seekbar accordingly. Using threads in Android, though, requires precautions, as we cannot rely on the classical Java threading model. Indeed, you should remember that the main thread of the application (also called *UI tread*) is responsible of managing all the widget of the UI, and that the Android UI toolkit is not thread-safe. So, you must not manipulate your UI from a worker thread, you must do all manipulation to your user interface from the UI thread[4]. There are simply two rules to Android's single-thread model:

[4]More info here

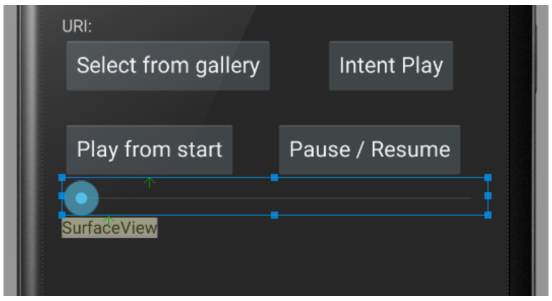


Figure 9: The SeekBar in the graphical layout.

1. Do not block the UI thread
2. Do not access the Android UI toolkit from outside the UI thread

For these reasons we will rather use an `ASyncTask` to update the seekbar. The main idea is to have the task always running when the application is active and checking the current state of the media player to update the seekbar accordingly[5].

- create a new inner class `BarUpdaterTask` that extends `ASyncTask`. Android Studio automatically generates the code stub for the method `doInBackground()` to implement. We also need to override the method `onProgressUpdate()` that updates the SeekBar widget.
- the implementation of `doInBackground()` should be a loop that calls the method `publishProgress()` to update the widget and then it sleeps for a given amount of time (*e.g.* 50 ms). The loop should end whenever `isCancelled()` becomes true.
- the implementation of `onProgressUpdate()` should just retrieve the current position from `MediaPlayer` and update the `SeekBar` widget through its `setProgress()` method.
- Finally, create a private attribute of the class `BarUpdaterTask` in the main class, start the task through its `execute()` method when the activity is started or brought back to the foreground (`onResume()`) and cancel it whenever it is paused (`onResume()`).
- Run and test the application.

2.4 Managing different orientations

So far our application runs just fine. Try to run the application on the emulator and press Ctrl + F12. The virtual device has switched to the landscape mode (Ctrl + F11 to get back). The result is quite disappointing, though: the control buttons take the whole screen and there is no space for displaying the video. We can then redefine a more optimized layout for the landscape mode. As we have seen in classes, Android keeps the resources and the visual elements of the application separated from the source code. Therefore, to fix the orientation problem we just have to redefine a proper layout file for the landscape mode (see Figure 11):

- open the layout file in Design Mode, click on the first combo box of the toolbar menu, and choose “Create Landscape Variation” (see Figure 10). A new layout with the same name is created in `res/layout-land`. This layout is loaded every time the device switch to the Landscape mode.

[5]Note that this may be not the optimal solution, we could start a task every time the video is started or resumed from pause and cancel it when the video is paused. For this tp we will implement a simpler solution.

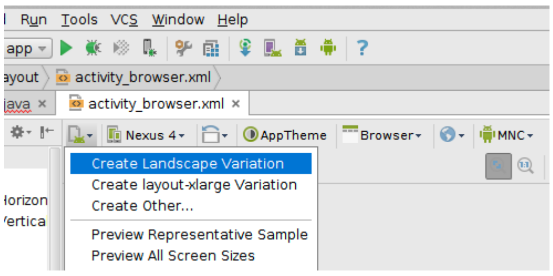

Run Tools VCS Window Help
app
ayout activity_browser.xml
java activity_browser.xml
Nexus 4 AppTheme Browser MNC
Create Landscape Variation
Create layout-xlarge Variation
Create Other...
Preview Representative Sample
Preview All Screen Sizes
Horizon
Vertica

Figure 10: How to generate a landscape variation for the portrait layout.

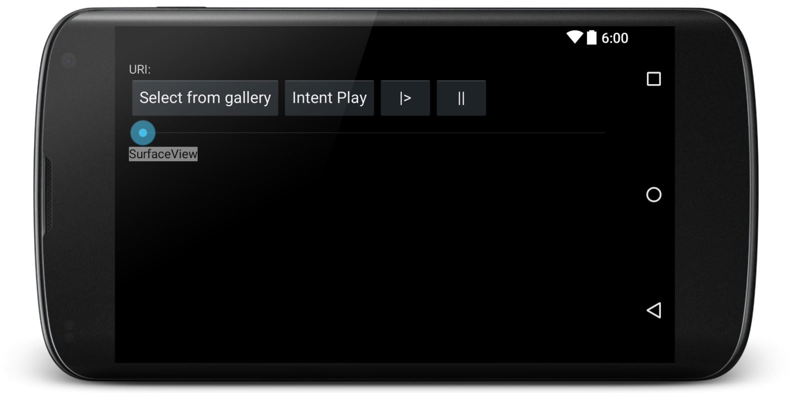


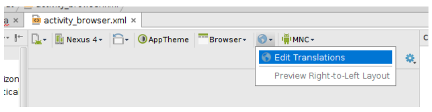
Figure 11: The landscape layout of the Video Player.

- create a new XML file with the same name as the original one; in `Available qualifiers` choose `Orientation` and the `Landscape` mode. A new file `res/layout-land/` is created, containing the layout for the landscape orientation;
- Now you can redefine the position and the size of the buttons as well as the text they display to save space and maximize the space for the video. In Figure 11 you can see an example of a possible layout. You must use the same ID for each component, while you can change the text using other strings to be placed as usual in `strings.xml`;
- to gain more space you can remove the action bar on top of the screen with the name of the application and the icon: change the application style to `Theme.Holo.NoActionBar` using the Style button of the toolbar menu of the Design Mode;
- Run again the application and see how it works in landscape mode.

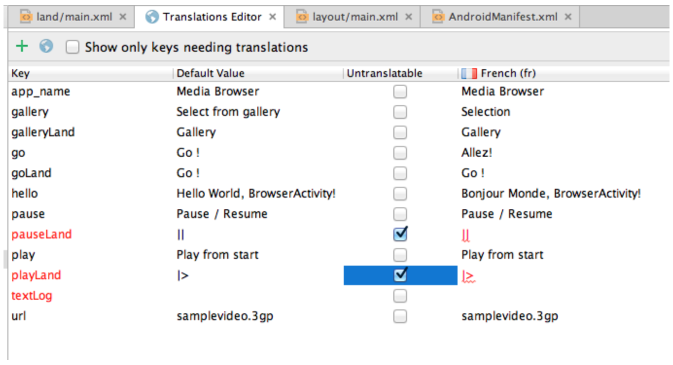
2.5 Managing internationalization and localization

Finally, to create a best-seller application it's important to add support for internationalization so that you can reach many markets and many more potential customers:

- open the layout file in Design Mode and click on the button with a “world” icon and select “Edit Translations”. This will open the translation editor (see Figure 12.a).
- In the translation editor click again on the button with a “world” icon and select the French country code `fr` (or `en` if you already used French for the text of the GUI).



(a)



Key	Default Value	Untranslatable	French (fr)
app_name	Media Browser	☐	Media Browser
gallery	Select from gallery	☐	Selection
galleryLand	Gallery	☐	Gallery
go	Go !	☐	Allez!
goLand	Go !	☐	Go !
hello	Hello World, BrowserActivity!	☐	Bonjour Monde, BrowserActivity!
pause	Pause / Resume	☐	Pause / Resume
pauseLand	||	✓	||
play	Play from start	☐	Play from start
playLand	|>	✓	|>
textLog		☐	
url	samplevideo.3gp	☐	samplevideo.3gp

(b)

Figure 12: The steps for managing the localization of the application.

- Give the translation for the new language for all the string. Note that you can check a box in case the string is not translatable or you do not want to translate it (in this case, *e.g.*, for the button play and pause which are not exactly text, see Figure 12).
- You should notice that a new file `string.xml` has been added to the `res/value` folder.
- Go to the emulator and change the global settings to the French language: now your application automatically loads the French version. This is a good way to keep the visual representation of the application separated from the source code.