file:// scheme is now not allowed to be attached with Intent on targetSdkVersion 24 (Android Nougat). And here is the solution.

Posted on 23 Jul 2016 15:11 | 32099 reads | 0 shares

Android Nougat is almost be publicly released. And as an Android developer, we need to prepare ourself to adjust **targetSdkVersion** to the latest one, 24, to let everything works perfectly on the newest release of Android.

And as always, everytime we adjust targetSdkVersion, we need to check and make sure that every single part of our code works perfectly fine. If you just simply change the number, I could say that your application is taking a high risk of crashing or malfunction. In this case, when you change your app's targetSdkVersion to 24, we need to check that every single function works flawlessly on Android Nougat (24).

And this is one of the checklist you need to mark done before releasing your new version. There is one big security change on Android N like quoted below:

> *Passing `file://` URIs outside the package domain may leave the receiver with an unaccessible path. Therefore, attempts to pass a `file://` URI trigger a `FileUriExposedException`. The recommended way to share the content of a private file is using the `FileProvider`.*

Summarily, `file://` is not allowed to attach with Intent anymore or it will throw `FileUriExposedException` which may cause your app crash immediately called.

This blog will talk about this issue and also about the solution how to make it work on Android N.

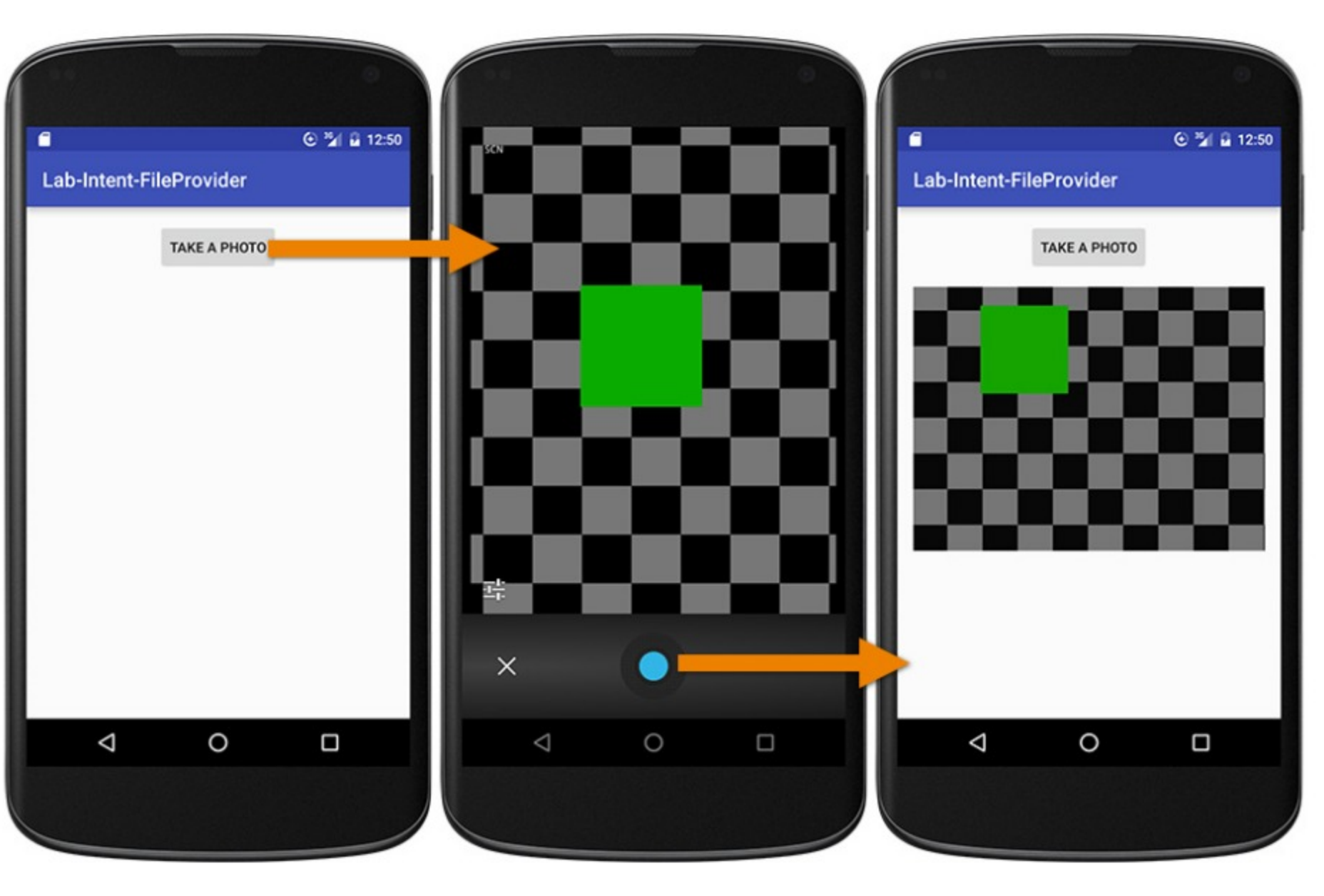
Real example with a crashing problem

You may be curious which situation that can really cause the problem. So to make it be easy to you all, let me show you a real usage example that causes crashing. **The easiest example is the way we take a photo through Intent with `ACTION_IMAGE_CAPTURE` type.** Previously we just pass the target file path with `file://` format as an Intent extra which works fine on Android Pre-N but will just simply crash on Android N and above.

Here is the code. Please note that you can find and download it from GitHub.

```
@RuntimePermissions
public class MainActivity extends AppCompatActivity implements View.OnClickListener {

    private static final int REQUEST_TAKE_PHOTO = 1;

    Button btnTakePhoto;
    ImageView ivPreview;

    String mCurrentPhotoPath;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        initInstances();
    }

    private void initInstances() {
        btnTakePhoto = (Button) findViewById(R.id.btnTakePhoto);
        ivPreview = (ImageView) findViewById(R.id.ivPreview);

        btnTakePhoto.setOnClickListener(this);
    }

    /////////////////////
    // OnClickListener //
    /////////////////////

    @Override
    public void onClick(View view) {
        if (view == btnTakePhoto) {
            MainActivityPermissionsDispatcher.startCameraWithCheck(this);
        }
    }

    ////////////
    // Camera //
    ////////////

    @NeedsPermission(Manifest.permission.WRITE_EXTERNAL_STORAGE)
    void startCamera() {
        try {
            dispatchTakePictureIntent();
        } catch (IOException e) {
        }
    }

    @OnShowRationale(Manifest.permission.WRITE_EXTERNAL_STORAGE)
    void showRationaleForCamera(final PermissionRequest request) {
        new AlertDialog.Builder(this)
                .setMessage("Access to External Storage is required")
                .setPositiveButton("Allow", new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialogInterface, int i) {
                        request.proceed();
                    }
                })
                .setNegativeButton("Deny", new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialogInterface, int i) {
                        request.cancel();
                    }
                })
                .show();
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);
        if (requestCode == REQUEST_TAKE_PHOTO && resultCode == RESULT_OK) {
            // Show the thumbnail on ImageView
            Uri imageUri = Uri.parse(mCurrentPhotoPath);
            File file = new File(imageUri.getPath());
            try {
                InputStream ims = new FileInputStream(file);
                ivPreview.setImageBitmap(BitmapFactory.decodeStream(ims));
            } catch (FileNotFoundException e) {
                return;
            }

            // ScanFile so it will be appeared on Gallery
            MediaScannerConnection.scanFile(MainActivity.this,
                    new String[]{imageUri.getPath()}, null,
                    new MediaScannerConnection.OnScanCompletedListener() {
                        public void onScanCompleted(String path, Uri uri) {
                        }
                    });
        }
    }

    @Override
    public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults) {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults);
        MainActivityPermissionsDispatcher.onRequestPermissionsResult(this, requestCode, grantResults);
    }

    private File createImageFile() throws IOException {
        // Create an image file name
        String timeStamp = new SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date());
        String imageFileName = "JPEG_" + timeStamp + "_";
        File storageDir = new File(Environment.getExternalStoragePublicDirectory(
                Environment.DIRECTORY_DCIM), "Camera");
        File image = File.createTempFile(
                imageFileName,  /* prefix */
                ".jpg",         /* suffix */
                storageDir      /* directory */
        );

        // Save a file: path for use with ACTION_VIEW intents
        mCurrentPhotoPath = "file:" + image.getAbsolutePath();
        return image;
    }

    private void dispatchTakePictureIntent() throws IOException {
        Intent takePictureIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
        // Ensure that there's a camera activity to handle the intent
        if (takePictureIntent.resolveActivity(getPackageManager()) != null) {
            // Create the File where the photo should go
            File photoFile = null;
            try {
                photoFile = createImageFile();
            } catch (IOException ex) {
                // Error occurred while creating the File
                return;
            }
            // Continue only if the File was successfully created
            if (photoFile != null) {
                Uri photoURI = Uri.fromFile(createImageFile());
                takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT, photoURI);
                startActivityForResult(takePictureIntent, REQUEST_TAKE_PHOTO);
            }
        }
    }
}
```

When code above is run, there will be a Button shown on screen. Once Button is clicked, camera app will be launched to let you take a photo. After everything is done, the taken photo will be shown on the ImageView as a result.

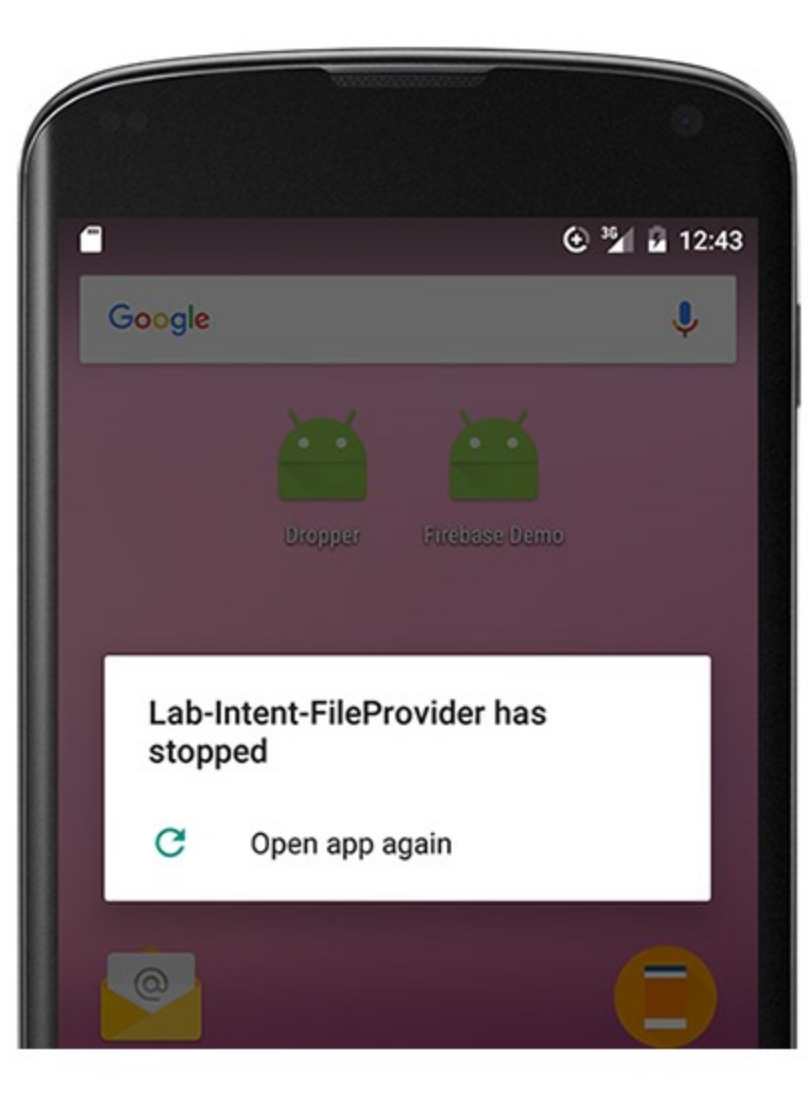


What the code does is quite straightforward. Generated file path pointed to DCIM folder under External Storage will be sent to camera app in `file://` format and will be used as file path of a photo taken.

Code above works fine even on Android Nougat since targetSdkVersion is still be 23. Now let's change it to 24.

```
android {
    ...
    defaultConfig {
        ...
        targetSdkVersion 24
    }
}
```

Here is the result. It still works on Android Pre-N but it appears to cause crashing on Android Nougat like this:



And here is the stacktrace.

```
FATAL EXCEPTION: main
    Process: com.inthecheesefactory.lab.intent_fileprovider, PID: 28905
    android.os.FileUriExposedException: file:///storage/emulated/0/DCIM/Camera/JPEG_20160723_124304_642070113.jpg exposed beyond a
    at android.os.StrictMode.onFileUriExposed(StrictMode.java:1799)
    at android.net.Uri.checkFileUriExposed(Uri.java:2346)
    at android.content.ClipData.prepareToLeaveProcess(ClipData.java:832)
    ...
```

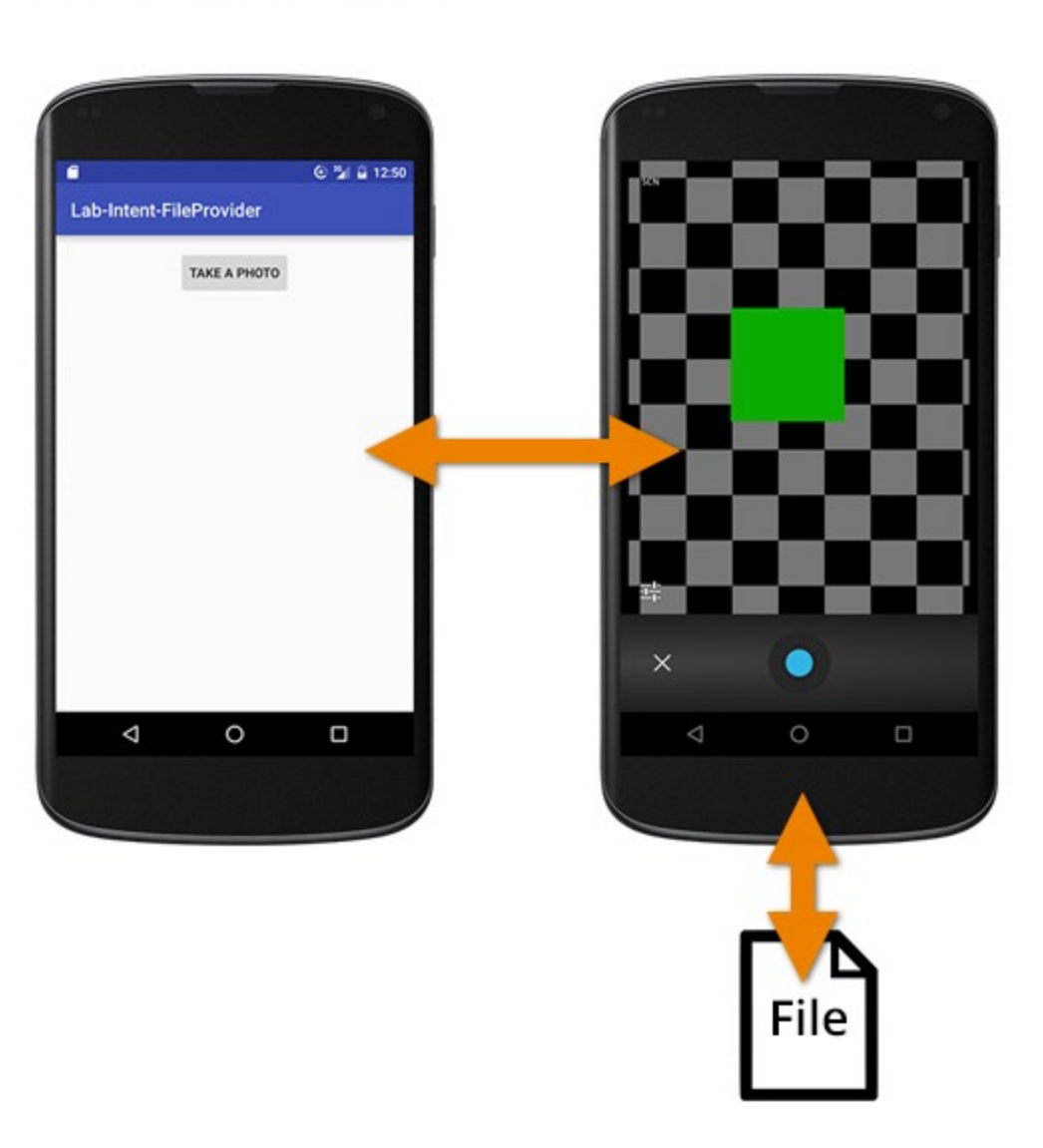
The reason is quite obvious. `file://` is not allowed as an attached URI in Intent or `FileUriExposedException` would be thrown.

And this is a big issue that you have to make sure that all code related to this case has already been fixed before releasing a new version with targetSdkVersion 24 or your app may crash on some of your user's device.

Why Nougat does not allow passing file:// with Intent anymore?

You may be curious why Android team decide to change this behavior. Actually there is a good reason behind.

If file path is sent to the target application (Camera app in this case), file will be fully accessed through the Camera app's process not the sender one.

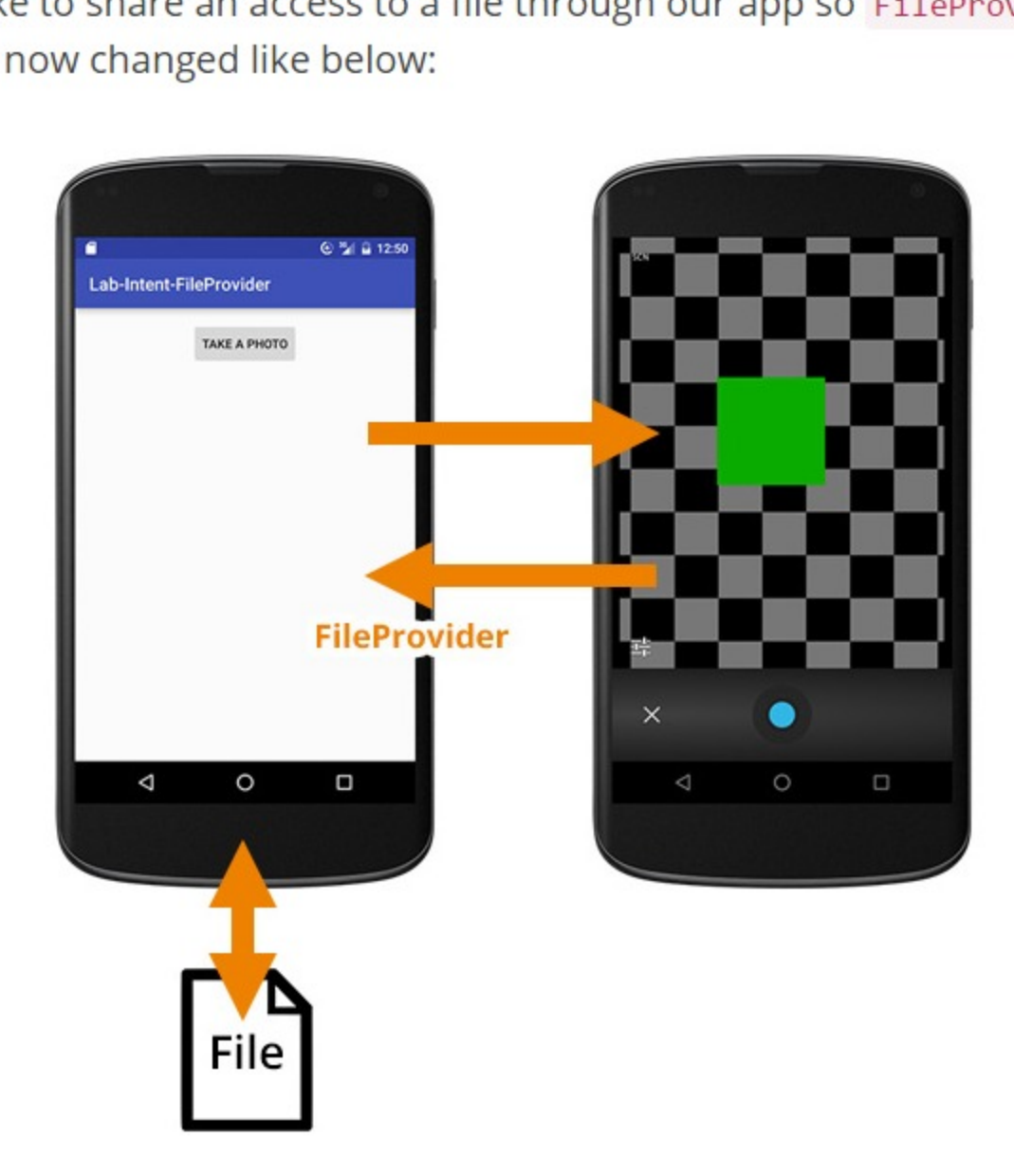


But let's consider thoroughly, actually Camera is launched by our application to take a photo and save as a file on our app's behalf. So the access right to that file should be our app's not Camera's. Every operation did with the file should be done through our application not by Camera app itself.

And that's why `file://` is now prohibited on targetSdkVersion 24 to force every developer to do this task in the proper way.

Solution

So if `file://` is not allowed anymore, which approach should we go for? The answer is we should send the URI through `content://` scheme instead which is the URI scheme for **Content Provider**. In this case, we would like to share an access to a file through our app so `FileProvider` is needed to be implemented. Flow is now changed like below:



And now, with FileProvider, file operation would be done through our app process like it supposes to be !

It is quite easy to implement FileProvider on your application. First you need to add a FileProvider `<provider>` tag in `AndroidManifest.xml` under `<application>` tag like below:

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    ...
    <application
        ...
        <provider
            android:name="android.support.v4.content.FileProvider"
            android:authorities="${applicationId}.provider"
            android:exported="false"
            android:grantUriPermissions="true">
            <meta-data
                android:name="android.support.FILE_PROVIDER_PATHS"
                android:resource="@xml/provider_paths"/>
        </provider>
    </application>
</manifest>
```

And then create a `provider_paths.xml` file in `xml` folder under `res` folder. Folder may be needed to create if it doesn't exist.

The content of the file is shown below. It describes that we would like to share access to the External Storage at root folder **(path=".")** with the name **external_files**.

res/xml/provider_paths.xml

```
<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android="http://schemas.android.com/apk/res/android">
    <external-path name="external_files" path="."/>
</paths>
```

Done! FileProvider is now declared and be ready to use.

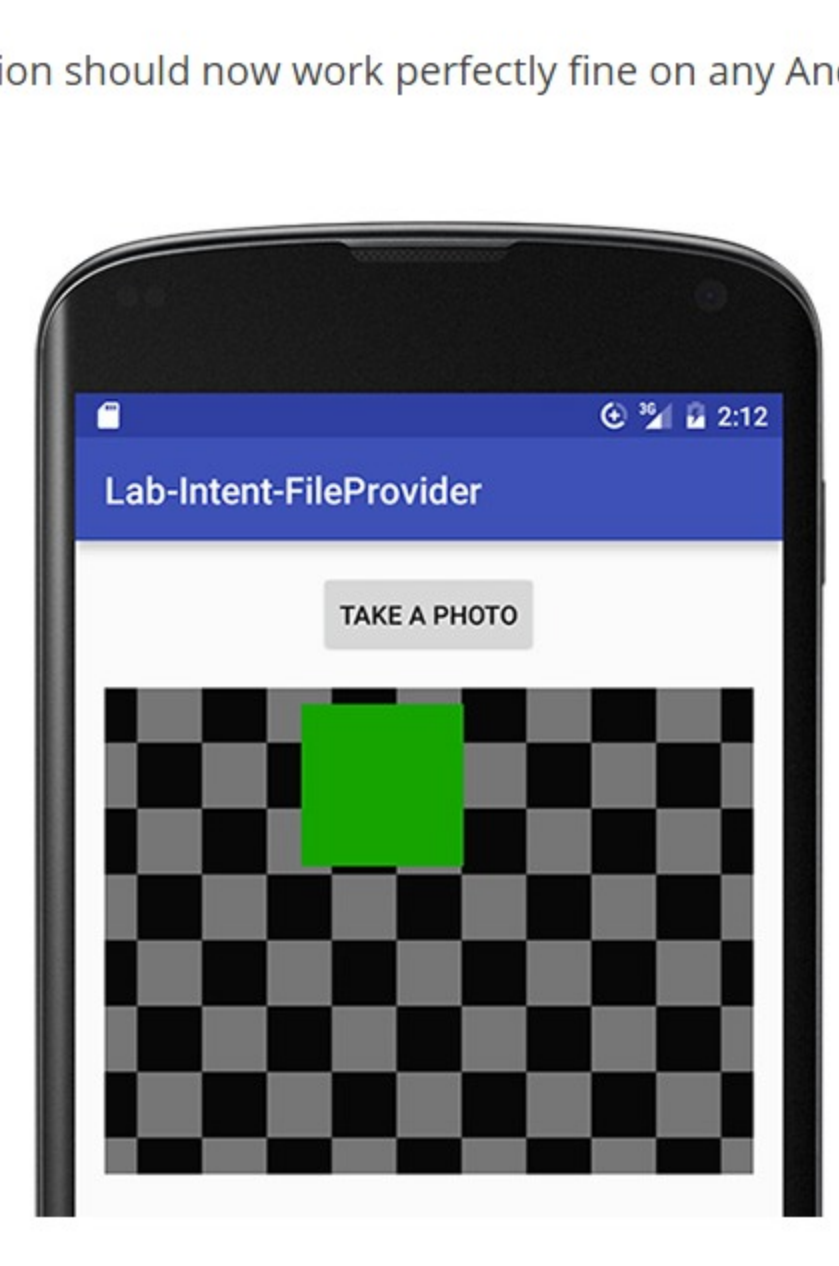
The final step is to change the line of code below in `MainActivity.java`

```
Uri photoURI = Uri.fromFile(createImageFile());
```

to

```
Uri photoURI = FileProvider.getUriForFile(MainActivity.this,
        BuildConfig.APPLICATION_ID + ".provider",
        createImageFile());
```

And .... done ! Your application should now work perfectly fine on any Android version including Android Nougat. Yah !



How about the existed app launched previously?

As you can see from the experiment result above. This behavior will happen only when you change your app's targetSdkVersion to 24 or above. So if your previously launched application was set the targetSdkVersion to 23 or lower, it supposes not to be a problem even on Android Nougat. `file://` should still works perfectly fine.

Anyway, to match the Android Best Practice, when there is a new API Level, we better always change targetSdkVersion to the latest one for the best user experience. =)

Author: **nuuneoi** (Android GDE, CTO & CEO at The Cheese Factory)
A full-stack developer with more than 6 years experience on Android Application Development and more than 12 years in Mobile Application Development industry. Also has skill in Infrastucture, Service Side, Design, UI&UX, Hardware, Optimization, Cooking, Photographing, Blogging, Training, Public Speaking and do love to share things to people in the world!

20 Comments Sort by Oldest

Add a comment...

Root Essentials
Nice article you have published! I still wonder what I am missing when I use FileProvider. The secondary application does not find the file on handover of the path. When I try on a root directory, I get error like: java.lang.IllegalArgumentException: Failed to find configured root that contains /system/build.prop. Any idea how to proceed?
Like · Reply · Jul 24, 2016 7:53pm

Jiaxin Ji · 山东大学
good as ever
Like · Reply · Aug 15, 2016 1:28am

Sascha Günther · Stadt Zwickau, Sachsen, Germany
Thanks very much. You saved my day.
Like · Reply · Aug 26, 2016 10:19am

Jonas Prapuolenis · Works at Intermedix Corporation
Shouldn't the provider be exported="true"? I got the following exception on one Android 4.4.3 device while testing: java.lang.SecurityException: Permission Denial: opening provider android.support.v4.content.FileProvider from ProcessRecord{41f05670 5969:com.android.camera2/u0a4} (pid=5969, uid=10004) that is not exported from uid 10160
Like · Reply · Sep 2, 2016 8:01pm

Jonas Prapuolenis · Works at Intermedix Corporation
Dang... Photo app can't use my FileProvider on Android 4.4.3, but I also can't export it: Caused by: java.lang.SecurityException: Provider must not be exported

So another if(api >=24){useFileProvider()} else { useFileUri() }....
Like · Reply · 1 · Sep 2, 2016 8:05pm