eecs441§3&4

Chatter with Images Kotlin

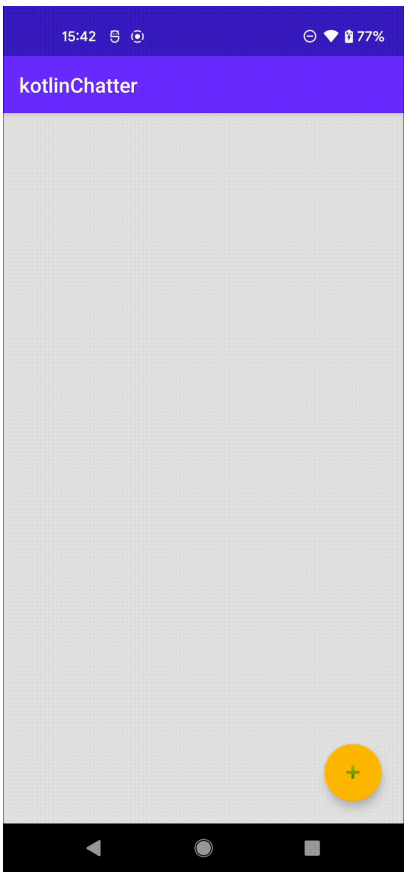
DUE Wed, 09/29, 2 pm

Welcome to the front end for Lab 1! In this lab, we will be using Android's common Camera component to add and manipulate images and videos in `Chatter`. We assume that you have completed the back end server setup, which is described in a separate spec.

Gif demo

Post an image and a video:

> Right click on the gif and open in a new tab to get a full-size view. To view the gif again, please hit refresh on the browser (in the new tab where the gif is opened).



Uploading images and videos

Images and videos can be uploaded to the server either by picking one from the device's photo album or by taking a picture/video with the device's camera. Images will be downloaded and displayed with given `chatt`s. On the posting screen, we will want a button to access the album, one for taking photo, another

for recording video, and a preview of the image to be posted. On the Main screen showing the `chatt` timeline, we will want posted images to show up alongside their corresponding `chatt` s and a button to play back any posted video. Let's get started.

Preparing your GitHub repo

- On your laptop, navigate to `YOUR_LABSFOLDER/`
- Create a zip of your `lab0` folder
- Rename your `lab0` folder **lab1**
- Push your local `YOUR_LABSFOLDER/` repo to GitHub and make sure there're no git issues

In Android Studio's `File > Open` choose `YOUR_LABSFOLDER/lab1/kotlinChatter` .

Declaring dependencies

We will be using two third-party SDKs in this lab: Coil, to help with image downloading, and OkHttp, to help with `multipart/form-data` upload. We will also be using the Android `ActivityResultContracts` API to return results from one activity to another.

In Android Studio, navigate to `File > Project Structure > Modules > Properties` . Set both `Source Compatibility` and `Target Compatibility` to **1.8 (Java 8)**. Coil depends on it.

Add the following lines to your app-level gradle file, `/Gradle Scripts/build.gradle (Module: kotlinChatter.app)` :

```
dependencies {
    ...
    implementation 'io.coil-kt:coil:1.3.2'
    implementation 'com.squareup.okhttp3:okhttp:5.0.0-alpha.2'
}
```

Bring up the `Project Structure` window (`⌘;` on the Mac, `Ctl-Alt-Shift-s` on Windows). If the last item on the left pane, `Suggestions` , shows a number next to it, click on the item and click `Update` on all of the suggested updates, click `Apply` , and then `OK` .

Adding camera feature

First things first, our application will make use of the camera feature. Navigate to your `AndroidManifests.xml` file and add the following inside the `<manifest...> ... </manifest>` block.

```
<uses-feature
    android:name="android.hardware.camera2"
    android:required="false" />
```

> There was also an `android.hardware.camera` , but it has been deprecated and is not compatible with `camera2` , so be careful which one you use.

> We set `android:required` to `false` to let users whose devices don't have a camera to continue to use the app. However, we will have to manually check later if there's a camera and disable picture and video taking if there isn't one.

Requesting permissions

Next we must declare we will be asking user's permission to access the device's camera, mic, and image gallery. Add these permission tags to your app's `AndroidManifest.xml` file. Find `android.permisssion.INTERNET` and add the following lines right below it:

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.RECORD_AUDIO" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

Without these permission tags, we wouldn't be able to prompt the user for permission later on. We're requesting permission to read external storage, so that we can select images from the photo album and obtain the result of image cropping, which will be performed by an external Activity.

We also need to declare that we will be requesting for image cropping capability from external Activities. Add the following to your `AndroidManifest.xml` , for example before the `<application></application>` block:

```
<queries>
    <intent>
        <action android:name="com.android.camera.action.CROP" />
        <data android:mimeType="image/*" />
    </intent>
</queries>
```

UI for posting

In lab0, we created the UI graphically using Android Studio's Layout Editor. The resulting UI is stored in XML files (in `/app/res/layout`). In this lab, we learn how to build UI by editing these XML files. Add the following strings to your `/app/res/values/strings.xml` within the `<resources></resources>` block:

```
<string name="album">Album</string>
<string name="camera">Camera</string>
<string name="video">Video</string>
<string name="preview">Image</string>
<string name="chattImage">chattImage</string>
<string name="videoView">videoView</string>
```

We now work on the `PostActivity` screen: we want to add an image preview and three buttons. An album button to let user select either an image or a video clip from their album. Invoking it twice allows user to attach both an image and a video to a `chatt` . A camera button for taking picture. A video button for recording video. And an image view to let user preview the image to be posted (photo only, no video).

Navigate to your `/app/res/layout/activity_post.xml`, and add the following code **inside**** your `androidx.constraintlayout.widget.ConstraintLayout` block, **below** the `<EditText ... />` block:

```
    <ImageButton
        android:id="@+id/albumButton"
        android:contentDescription="@string/album"
        android:layout_width="40dp"
        android:layout_height="40dp"
        android:layout_marginEnd="12dp"
        android:layout_marginTop="16dp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/messageTextView"
        android:background="@drawable/border"
        app:srcCompat="@android:drawable/ic_menu_gallery" />
```

This adds a UI element of type `ImageButton`. The button is given ID `albumButton` with `contentDescription` as stored in the string named "album" in the '/res/values/strings.xml` file.

Next we specify the four things about the UI element that `ConstraintLayout` needs to position it:

1. width: 40dp
2. height: 40dp
3. the x-coordinate and
4. y-coordinate of one of the element's corners

The "corner" we pick to specify is the top-right(end) corner of the `ImageButton`. Here we specify that the end (right) edge of the button should be 12dp from the end edge of the parent container and the top edge of the button should be 16dp from the bottom edge of another UI element whose ID is `messageTextView` (recall that we added this UI element in the first lab).

After specifying the layout of the UI element, we specify that its background should be the image stored in a file called `border.xml` in the `/app/res/drawable/` folder. In Android Studio, right click on the `/app/res/drawable/` folder and select `New > Drawable Resource File`. In the popup dialog box, enter `File name:` "border" and click `OK`. Replace the content of the newly created `border.xml` file with the following:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle">
    <gradient android:startColor="#F3F3F3"
        android:endColor="#D9D9D9"
        android:angle="270" />
    <corners android:radius="8dp" />
    <stroke android:width="3dp" android:color="#C0C0C0" />
</shape>
```

Finally, we specify that the `ImageButton` will have a system icon called `ic_menu_gallery`. The `android:` tag in front of the `drawable` tells Android Studio to obtain the icon from the Android system library.

We're done adding the `albumButton`. Next add another `ImageButton` to `/res/layout/activity_post`:

- give it the ID `cameraButton`
- set its `contentDescription` to be the string stored with the "camera" name in `/res/values/strings.xml`
- set both its width and height to 40dp
- set its top-end corner such that its top edge is 16dp from the bottom edge of `messageTextView` same as the `albumButton`
- **but** its end (right) edge is 12dp from the start (left) edge of `albumButton`
- let it use the same `border` background as `albumButton`, and
- let it have `ic_menu_camera` as its icon

Add one more `ImageButton`:

- give it the ID `videoButton`
- set its `contentDescription` to be the string stored with the "video" name in `/res/values/strings.xml`
- set both its width and height to 40dp
- set its top-end corner such that its top edge is 16dp from the bottom edge of `messageTextView` same as the `albumButton`
- **but** its end (right) edge is 12dp from the start (left) edge of `cameraButton`
- for its background, we'll use a system background: `"@android:color/transparent"`
- for its icon, we'll use a system icon: `presence_video_online`
- since the icon is very small, we add one more tag **inside** this `<ImageButton .../>` block to scale it up: `android:scaleType="fitCenter"`

One last thing we need to add to `PostActivity` screen is a preview of the image (photo only, no video) to be posted. Instead of an `ImageButton`, this time we add an `ImageView`:

- give it the ID `previewImage`
- set its `contentDescription` to be the string stored with the "preview" name in `/res/values/strings.xml`
- set both its width and height to **"wrap_content"**
- set its top-start corner such that its top edge is 12dp from the bottom edge of `messageTextView` same as the `albumButton`
- **but** its start (left) edge is 0dp from the start (left) edge of `messageTextView` (so it's flushed left with `messageTextView`)

Connect UI with code

Now that we've updated our `PostActivity` layout, let's add some functionalities to our UI. Start by adding the properties below to your `PostActivity` class.

```
private lateinit var forCropResult: ActivityResultLauncher<Intent>

private var imageUri: Uri? = null
private var videoUri: Uri? = null
```

> `URI` stands for Uniform Resource Identifier and is a standard, hierarchical way to name things on the Internet as defined in RFC2396. It is different from URL in that it doesn't necessarily tell you how to

> locate the thing.

Next, follow up on the permission tags added to `AndroidManifest.xml` above with code in `onCreate()` to prompt user for access permissions. We will be using one of Android's standard-activity `ActivityResultContracts` to request permissions to access the camera, mic, and external storage. Add the following to your `onCreate()`:

```
registerForActivityResult(ActivityResultContracts.RequestMultiplePermissions()) { results ->
    results.forEach {
        if (!it.value) {
            toast("${it.key} access denied")
            finish()
        }
    }
}.launch(arrayOf(Manifest.permission.CAMERA,
        Manifest.permission.RECORD_AUDIO,
        Manifest.permission.READ_EXTERNAL_STORAGE))
```

We did three things in the above code. First we created a "contract" that informs Android that a certain Activity will be started and the Activity will be expecting input of a certain type and will be returning output of other certain type. This ensures the type safety of starting an Activity for results. In this case, we specified that the Activity we want to start is to request multiple permissions, which is a standard Activity for which Android already provides a canned contract with baked-in input/output types.

The second thing we did after creating the "contract" was to register it with the Android OS by calling `registerForActivityResult()`. As part of the registration process, we provided a callback to handle results from starting the Activity, in the form of a trailing lambda expression. The callback handler will examine the result of each permission request. If any of the permission is denied (`it.value == false`), for the sake of expediency, we will simply inform the user which permission (`it.key`) has been denied with a `toast()`, end `PostActivity`, and return to `MainActivity`. In a real app, you may want to be less draconian and let user continue to post text messages.

Since activities can be and are destroyed and re-created, for example everytime the screen orientation changes, the registration of activity result contracts **must** be done in an Activity's `onCreate()`. This way, every time the Activity is re-created, the contract is re-registered.

> ℹ You must register a contract for each Activity that you want to start from the current Activity if you expect results to be returned from the launched Activity. Contract registrations must all be done in the current Activity's `onCreate()`.

The call to `ActivityResultContracts()` returns a contract that we can store in a local variable. In this case, since we have no further use of the contract, we didn't store it in a variable. Instead, we use it directly in the call to `registerForActivityResult()`. If we had stored the contract in a local variable first, and name the argument to the lambda expression provided to `forEach`, `result`, the code above would be the equivalent of:

```
        val contract = ActivityResultContracts.RequestMultiplePermissions()
        registerForActivityResult(contract) { results ->
            results.forEach { result ->
                if (!result.value) {
                    toast("${result.key} access denied")
                    finish()
                }
            }
        }
```

The third thing we did in the above code, was to launch the registered contract to ask access permission to the camera, mic, and external storage. The call to `registerForActivityResult()` returns a registration handler that we are again not storing in a local variable, but have instead directly called its `launch()` method. If we had stored both the contract and the registration handler in local variables, the code above would be the equivalent of:

```
        val contract = ActivityResultContracts.RequestMultiplePermissions()
        val launcher = registerForActivityResult(contract) { results ->
            results.forEach { result ->
                if (!result.value) {
                    toast("${result.key} access denied")
                    finish()
                }
            }
        }
        launcher.launch(arrayOf(Manifest.permission.CAMERA,
            Manifest.permission.RECORD_AUDIO,
            Manifest.permission.READ_EXTERNAL_STORAGE)
```

Notice that we use a `Toast` to inform the user if permission has been denied. A `Toast` is a small pop-up that appears on screen. Toasts can be very helpful while debugging and notify the user of their current state in the app. Instead of using `Toast` directly however, we have added a `toast()` extension to the application `Context`. The extension allows us to use `Toast` with some boiler-plate arguments pre-set. By attaching the extension to the application `Context`, we can use it with any `Activity`, not just `PostActivity`.

We'll collect all the extensions we'll be using into one file. Create a new Kotlin File (**not** `Class`) called `Extensions.kt` and put the following code in it:

▶ File vs. Class

```
fun Context.toast(message: String, short: Boolean = true) {
    Toast.makeText(this, message, if (short) Toast.LENGTH_SHORT else Toast.LENGTH_LONG).show()
}
```

> Recall that if Android Studio prompts you with multiple possible imports and you're not sure which one to import, you can consult the Appendix for a full list of imports each Activity needs.

Now, let's add the function to pick an image and/or video clip from the photo album. We will again be using Android standard-activity `ActivityResultContracts` to perform this activity. Staying in `onCreate()`, add the following code after the code to request permissions:

```
val cropIntent = initCropIntent()
val forPickedResult =
    registerForActivityResult(ActivityResultContracts.GetContent(), fun(uri: Uri?) {
        uri?.let {
            if (it.toString().contains("video")) {
                videoUri = it
                view.videoButton.setImageResource(android.R.drawable.presence_video_busy)
            } else {
                val inStream = contentResolver.openInputStream(it) ?: return
                imageUri = mediaStoreAlloc("image/jpeg")
                imageUri?.let {
                    val outStream = contentResolver.openOutputStream(it) ?: return
                    val buffer = ByteArray(8192)
                    var read: Int
                    while (inStream.read(buffer).also{ read = it } != -1) {
                        outStream.write(buffer, 0, read)
                    }
                    outStream.flush()
                    outStream.close()
                    inStream.close()
                }
                doCrop(cropIntent)
            }
        } ?: run { Log.d("Pick media", "failed") }
    })
view.albumButton.setOnClickListener {
    forPickedResult.launch("*/*")
}
```

Aside from the much more complicated callback handler, which we'll walk through in a bit, there are four differences with the previous contract I'd like to draw your attention to.

First, instead of the canned `RequestMultiplePermissions` contract, we use the provided `GetContent` contract. Second, instead of providing `registerForActivityResult()` with a callback handler in the form of a trailing lambda expression, we're using an **anonymous function**. The advantage is that we can structurally `return` from an anonymous function without exiting the enclosing function. And we can do so without relying on a goto label. We execute return from the anonymous function a couple of times: when we fail to open an input or output stream. Third, instead of launching the registered contract directly, we did save it in a local variable, `forPickedResult`. And fourth, the reason we saved the registration result is to assign it to the `albumButton` so that it can be launched at a later time when the button is clicked.

To pick an image or video from the device's photo album or from Google Drive, we use the `GetContent` `ActivityResultContract`. To pick from on-device album, user must manually choose `Photos` from the menu. In registering a callback handler for this activity, instead of a lambda expression, we have used an **anonymous function** that takes a single argument of type nullable-Uri. If the `uri` is not null and contains the word "video", we store it in `videoUri` and change the `videoButton` icon (to red). If it's not a video uri, then it's an image uri. To allow user to crop and zoom the image, we need to make a copy first, then we call `doCrop()` before posting.

When we call `doCrop()`, the user will be redirected to a separate cropping activity. We rely on third-party cropping capability published on device to perform the cropping function. To subscribe to this external capability, we first create an external activity Intent using `initCropIntent()`. Here's the implementation of `initCropIntent()`. Put it outside your `onCreate()` function:

```
private fun initCropIntent(): Intent? {
    // Is there any published Activity on device to do image cropping?
    val intent = Intent("com.android.camera.action.CROP")
    intent.type = "image/*"
    val listofCroppers = packageManager.queryIntentActivities(intent, 0)
    // No image cropping Activity published
    if (listofCroppers.size == 0) {
        toast("Device does not support image cropping")
        return null
    }

    intent.component = ComponentName(
            listofCroppers[0].activityInfo.packageName,
            listofCroppers[0].activityInfo.name)

    // create a square crop box:
    intent.putExtra("outputX", 500)
            .putExtra("outputY", 500)
            .putExtra("aspectX", 1)
            .putExtra("aspectY", 1)
            // enable zoom and crop
            .putExtra("scale", true)
            .putExtra("crop", true)
            .putExtra("return-data", true)

    return intent
}
```

This function first searches for availability of external on-device Activity capable of cropping. If such an Activity exists, it creates an explicit intent to redirect the user to the image cropper, pre-setting the intent to include our desired cropping features.

And here's the `doCrop()` function to launch the external activity. Put it outside your `onCreate()` also:

```
private fun doCrop(intent: Intent?) {
    intent ?: run {
        imageUri?.let { view.previewImage.display(it) }
        return
    }

    imageUri?.let {
        intent.data = it
        forCropResult.launch(intent)
    }
}
```

The function `doCrop()` uses `forCropResult` to launch the external activity. The variable `forCropResult` contains the registered contract to start a generic Activity for result. Add the following initialization of this

variable **inside** your `onCreate()`:

```
        forCropResult =
                registerForActivityResult(ActivityResultContracts.StartActivityForResult()) { result
                    if (result.resultCode == Activity.RESULT_OK) {
                        result.data?.data.let {
                            imageUri?.run {
                                if (!toString().contains("ORIGINAL")) {
                                    // delete uncropped photo taken for posting
                                    contentResolver.delete(this, null, null)
                                }
                            }
                            imageUri = it
                            imageUri?.let { view.previewImage.display(it) }
                        }
                    } else {
                        Log.d("Crop", result.resultCode.toString())
                    }
                }
```

You can think of the `StartActivityForResult` contract as a "generic" contract that always returns the result in the form of a `resultCode` containing the value, `Activity.RESULT_OK`, `Activity.RESULT_CANCELLED`, or `Activity.RESULT_FIRST_USER`. Additional information resulting from the Activity will be put in `result.data`. If the Activity you want to start conforms to this return pattern, you can use the provided `StartActivityForResult` contract to launch it. (Most standard Android system activities conform to this result pattern, so you don't need to define your own custom contract to start most standard Android system activities.)

The callback handler we registered for the crop activity checks whether the uncropped original was taken by the camera or from an album. If it's taken by the camera, we can delete the original (otherwise, the app doesn't have permission to delete it). The cropped image is then displayed as a preview image. Image display is implemented as an extension function to the `ImageView` class. Add the following to your `Extensions.kt` file:

```
fun ImageView.display(uri: Uri) {
    setImageURI(uri)
    visibility = View.VISIBLE
}
```

When the user picks a photo, we allocate some scratch space in Android's `MediaStore` for use by the cropping function. To take picture and record video, we also need to allocate space in the `MediaStore` to store the picture/video. Put the following `mediaStoreAlloc()` function outside your `onCreate()`:

```
    private fun mediaStoreAlloc(mediaType: String): Uri? {
        val values = ContentValues()
        values.put(MediaStore.MediaColumns.MIME_TYPE, mediaType)
        values.put(MediaStore.MediaColumns.RELATIVE_PATH, Environment.DIRECTORY_PICTURES)

        return contentResolver.insert(
                if (mediaType.contains("video"))
```

```
                MediaStore.Video.Media.EXTERNAL_CONTENT_URI
            else
                MediaStore.Images.Media.EXTERNAL_CONTENT_URI,
            values)
    }
```

Back in `onCreate()`, we're done with picking photo or video from album. We now move on to adding functions to take a picture or a video clip using the camera. First, we check wehther the device has a camera. Add the following code after the initialization of the `forCropResult` variable **inside** your `onCreate()`:

```
        if (!packageManager.hasSystemFeature(PackageManager.FEATURE_CAMERA_ANY)) {
            toast("Device has no camera!")
            return
        }
```

If there's a camera, we set up the `ActivityResultContracts` to take picture. The name of the contract is, as expected, `TakePicture` (instead of `GetContent`, for example). Once the contract is created, register it with the following callback handler. We can provide the handler in the form of a trailing lambda in this case:

```
{ success ->
    if (success) {
        doCrop(cropIntent)
    } else {
        Log.d("TakePicture", "failed")
    }
}
```

If a picture is successfully taken, the handler calls `doCrop()` to let user edit the picture before posting.

Save the result of the contract registration in a local variable that we can then assign to the `cameraButton`. When the `cameraButton` is clicked, first initialize the class property `imageUri` with temporary storage space in the `MediaStore` for `mediaType = "image/jpeg"` and then launch the contract with `imageUri` as the launch argument.

Next set up the `videoButton` to record video similarly:

- use the standard activity for result contract called `TakeVideo`
- use the following callback handler when registering the contract:

```
{
  view.videoButton.setImageResource(android.R.drawable.presence_video_busy)
}
```

- When the `videoButton` is clicked, first initialize `videoUri` with `MediaStore` space for `mediaType = "video/mp4"`, then launch the contract with `videoUri` as the launch argument.

> Unlike picking media from the album, as of API Level 30, Android's camera API doesn't allow user to perform an image or video capture with one call, hence we need to have two different buttons making two different calls.

⚠ Be mindful that both Django and Nginx have an upper limit on client upload size. Be sure to adjust the upload threshold of **both** Nginx and Django to accommodate your target video size.

On Pixel 3, the default resolution of videos captured with `ACTION_VIDEO_CAPTURE` is 1960x1080, resulting in 3 MB of data for 3 secs of video.

An alternative to using `ACTION_VIDEO_CAPTURE` is to use the `MediaRecorder` which gives finer control over resolution and duration, should you be interested.

We can now test our app! Make sure that when you tap the `albumButton` you are able to choose an image from your photo gallery, zoom and crop, and then preview it. Similarly the `cameraButton` and `videoButton` should allow you to take picture and record video, respectively. You can't submit the `chatt` yet, we'll work on that next.

`submitChatt()`

Next, we work on `submitChatt()`. Unlike in the previous labs, we will send image and video to post, if any, as URIs outside of a `chatt` object. In addition to the content of a `chatt` to `postChatt()`, we also send along a completion handler in the form of a trailing lambda. The completion handler informs the user of success or failure in posting and closes the `PostActivity`. As with the first lab, the network transmission itself will be done asynchronously on a separate thread. Any UI-related task can **only** be done on the main/UI thread, hence we need to run `toast()` on the UI thread. And the asynchronous task needs a home to return to upon completion, which is why we put the `finish()` call in the completion handler.

```
fun submitChatt() {
    val chatt = Chatt(username = view.usernameTextView.text.toString(),
        message = view.messageTextView.text.toString())

    postChatt(applicationContext, chatt, imageUri, videoUri) { msg ->
        runOnUiThread {
            toast(msg)
        }
        finish()
    }
}
```

Before we leave `PostActivity` to work on `postChatt()` in `ChattStore`, we have one more thing to take care of: when there is a configuration change, such as a screen rotation, Android will destroy and re-create your current Activity.

> You can use `Log.d()` to verify that when you change screen orientation, your Activity's `onDestroy()` and then its `onCreate()` will be called.

To retain user's image and video data and the `enableSend` state across configuration changes, we need to add the following two methods to `PostActivity`:

```
override fun onSaveInstanceState(savedInstanceState: Bundle) {
    super.onSaveInstanceState(savedInstanceState)
```

```
        savedInstanceState.putParcelable("imageUri", imageUri)
        savedInstanceState.putParcelable("videoUri", videoUri)
        savedInstanceState.putBoolean("enableSend", enableSend)
    }

    override fun onRestoreInstanceState(savedInstanceState: Bundle) {
        super.onRestoreInstanceState(savedInstanceState)
        imageUri = savedInstanceState.getParcelable<Uri>("imageUri")
        imageUri?.let { view.previewImage.display(it) }
        videoUri = savedInstanceState.getParcelable<Uri>("videoUri")
        enableSend = savedInstanceState.getBoolean("enableSend")
    }
```

> You could also prevent screen orientation changes in an **Activity** with:
>
> ```
> setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_NOSENSOR)
> ```
>
> However, you'd need to do this for **every** activity you want to have locked orientation, there's no app-wide setting–though there are ways to automate this, see references.

`postChatt()`

We will use the OkHttp SDK to upload the image/video using `multipart/form-data` representation/encoding.

> ℹ When a web page has a form for user to fill out, such page usually has mutiple fields (e.g., name, address, networth, etc.), each comprising a separate part of the multi-part form. Data from these multiple parts of the form is encoded for sending by HTTP using the native `multipart/form-data` representation. One advantage of using this encoding instead of JSON is that binary data can be sent as is, not encoded into a string of printable characters. Since we don't have to encode the binary data into character string, we can also stream directly from file to network without having to first load the whole file into memory, allowing us to send much larger files. These are the two reasons we use the `multipart/form-data` encoding instead of JSON in this lab.

Add the `client` property to your `ChattStore` object and replace your `postChatt()` function in the `ChattStore` class with the following version:

```
    private val client = OkHttpClient()

    fun postChatt(context: Context, chatt: Chatt, imageUri: Uri?, videoUri: Uri?,
                  completion: (String) -> Unit) {
        val mpFD = MultipartBody.Builder().setType(MultipartBody.FORM)
                .addFormDataPart("username", chatt.username ?: "")
                .addFormDataPart("message", chatt.message ?: "")

        imageUri?.run {
            toFile(context)?.let {
                mpFD.addFormDataPart("image", "chattImage",
                    it.asRequestBody("image/jpeg".toMediaType()))
```

```
            } ?: context.toast("Unsupported image format")
        }

        videoUri?.run {
            toFile(context)?.let {
                mpFD.addFormDataPart("video", "chattVideo",
                        it.asRequestBody("video/mp4".toMediaType()))
            } ?: context.toast("Unsupported video format")
        }

        val request = Request.Builder()
                .url(serverUrl+"postimages/")
                .post(mpFD.build())
                .build()

        context.toast("Posting . . . wait for 'Chatt posted!'")

        client.newCall(request).enqueue(object : Callback {
            override fun onFailure(call: Call, e: IOException) {
                completion(e.localizedMessage ?: "Posting failed")
            }

            override fun onResponse(call: Call, response: Response) {
                if (response.isSuccessful) {
                    completion("Chatt posted!")
                }
            }
        })
    }
```

By declaring `OkHttpClient` a property of our `ChattStore` object, we create only a single instance of the client, as recommended by the `OkHttp` document, to improve performance.

The code constructs the "form" to be uploaded as comprising a part named "username" with the field containing the username (or the empty string if `null`). Next it appends a part named "message" constructed similarly. Then comes a part named "image" with data as referenced by `imageUri`. The image has been JPEG encoded. The string "chattImage" is how the data is tagged, it can be any string. The `MediaType()` documents the encoding of the data (though it doesn't seem to be used for anything). The last part is named "video", the data is not in memory, but rather must be retrieved from `videoUri`. Upon completion of upload, the response is reported to the user using the provided callback lambda.

To upload data directly from `MediaStore`, given its URI, we rely on the `toFile()` function, which we write as an extension to the `Uri` class. Add the following function to your `Extensions.kt` file:

```
fun Uri.toFile(context: Context): File? {

    if (!(authority == "media" || authority == "com.google.android.apps.photos.contentprovider")) {
        // for on-device media files only
        context.toast("Media file not on device")
        Log.d("Uri.toFile", authority.toString())
        return null
    }

    if (scheme.equals("content")) {
        var cursor: Cursor? = null
```

```
        try {
            cursor = context.getContentResolver().query(
                this, arrayOf("_data"),
                null, null, null
            )

            cursor?.run {
                moveToFirst()
                return File(getString(getColumnIndexOrThrow("_data")))
            }
        } finally {
            cursor?.close()
        }
    }
    return null
}
```

Depending on your upload bandwidth, uploading video can take a long time. Wait for the `Chatt posted!` toast to pop up before trying to refresh your app's time line to view the new `chatt`.

With the updated `PostActivity`, you can now take or select images and videos and send them to your `Chatter` back end! Since we haven't worked on image/video download, you can verify this by inspecting the content of your `chatts` table in the postgres database at the backend.

Let's move on to downloading images from your server to see them in your timeline.

Viewing posted images and videos

We are now at the final step! Getting the image and/or video from our server and showing them in the `chatts`!

In order to view image and video alongside `chatt`, we'll need to edit both the `Chatt` class, and the layout of a `Chatt` list item. Let's start with the layout. Open your `/app/res/layout/listitem_chatt.xml` file, and add the following `ImageView` and `ImageButton` to your `ConstraintLayout`.

First we add and `ImageView`:

- give it the ID `chattImage`
- set its `contentDescription` to be the string stored with the "chattImage" name in `/res/values/strings.xml`
- set both its width and height to "**wrap_content**"
- set its top-start corner such that its top edge is 8dp from the bottom edge of `messageTextView`
- and its start (left) edge is 8dp from the start (left) edge of `"parent"`

Next add an `ImageButton`:

- give it the ID `videoButton`
- set its `contentDescription` to be the string stored with the "video" name in `/res/values/strings.xml`

- set both its width to 60dp and its height to 50dp
- set its top-end corner such that its top edge is 12dp from the bottom edge of `timestampTextView`
- and its end (right) edge is 0dp from the end (right) edge of `timestampTextView`, so it is flushed right with `timestampTextView`
- for its background, we'll use the same transparent system background we used with the `videoButton` in the `PostActivity` screen
- for its icon, we'll use a system icon: `ic_menu_slideshow`

Display images and playing back video

We move on to the `Chatt` class. Navigate to your `Chatt.kt`. Modify the class to add properties to hold an image URL and a video URL, alongside existing properties from the previous lab.

```
class Chatt(var username: String? = null,
            var message: String? = null,
            var timestamp: String? = null,
            imageUrl: String? = null,
            videoUrl: String? = null) {
    var imageUrl: String? by ChattPropDelegate(imageUrl)
    var videoUrl: String? by ChattPropDelegate(videoUrl)
}
```

Both `imageUrl` and `videoUrl` use the `ChattPropDelegate` property delegate class. When there's no valid URL associated with `imageUrl` and `videoUrl`, we want the value of these properties to be `null` `String`s. Unfortunately an empty value in a JSON object will be encoded as `JSONObject.NULL` by the Android JSON library, which when typecast to `String` is given the value `"null"`, i.e., a string with the characters n-u-l-l inside. The `ChattPropDelegate` converts `"null"` and the empty string `""` into the `null` String. Add the following class to your `Chatt.kt` file:

```
class ChattPropDelegate private constructor ():
        ReadWriteProperty<Any?, String?> {
    private var _value: String? = null
        set(newValue) {
            newValue ?: run {
                field = null
                return
            }
            field = if (newValue == "null" || newValue.isEmpty()) null else newValue
        }

    constructor(initialValue: String?): this() { _value = initialValue }

    override fun getValue(thisRef: Any?, property: KProperty<*>) = _value
    override fun setValue(thisRef: Any?, property: KProperty<*>, value: String?) {
        _value = value
    }
}
```

The `ChattListAdapter` of lab0 recognizes only the username, message, and timestamp of each `Chatt`; let's change this to include image and video: if the poster has uploaded an image with a given `chatt`, it will

display the image corresponding to the `chatt` in `chattImage`. Similarly, if video was posted with a `chatt`, the `videoButton` will be shown and set up to play back the video clip when clicked. Add the following code to the end of, and **inside**, the `getItem(position)?.run{}` code block in the `getView()` method.

```
// show image
imageUrl?.let {
    listItemView.chattImage.setVisibility(View.VISIBLE)
    listItemView.chattImage.load(it) {
        crossfade(true)
        crossfade(1000)
    }
} ?: run {
    listItemView.chattImage.setVisibility(View.GONE)
    listItemView.chattImage.setImageBitmap(null)
}

videoUrl?.let {
    listItemView.videoButton.visibility = View.VISIBLE
    listItemView.videoButton.setOnClickListener { v: View ->
        if (v.id == R.id.videoButton) {
            val intent = Intent(context, VideoPlayActivity::class.java)
            intent.putExtra("VIDEO_URI", Uri.parse(it))
            context.startActivity(intent)
        }
    }
} ?: run {
    listItemView.videoButton.visibility = View.INVISIBLE
    listItemView.videoButton.setOnClickListener(null)
}
```

We use the image view `.load()` extension from the Coil SDK to download the image directly from its URL, also employing a crossfade effect to *mimic* progressive JPEG.

▶ Progressive JPEG

ℹ The visibility of a View on Android could be `VISIBLE`, `INVISIBLE`, or `GONE`. `INVISIBLE` means while the View is not visible, the space it would otherwise occupy will be left blank. `GONE` means the space it would otherwise occupy will be removed. By specifying the visibility of `chattImage` to `GONE` means if no image has been posted with a `chatt`, there won't be a big empty space between this and next `chatt`.

If a given `chatt` contains video, the `videoButton` will become visible, and when clicked, it will create a new intent to start the `VideoPlayActivity`, passing along the `videoUrl` associated with the `chatt`, tagged as `"VIDEO_URI"` in the intent. When there is no video, we explicitly turn the `videoButton` invisible and its `onClickListener` to `null`. Recall that `listItem`s are recycled and reused.

VideoPlayActivity

To view downloaded video, we'll create a new activity, `VideoPlayActivity` to play the video. Create this new activity from the main menu: `File > New > Activity > Empty Activity`. Name the activity `VideoPlayActivity` and click `Finish`.

Open the newly created `/res/layout/activity_video_play.xml` file and add a `VideoView` to your `ConstraintLayout`:

- give it the ID `videoView`
- set its `contentDescription` to be the string stored with the "videoView" name in `/res/values/strings.xml`
- set both its width and height to "**match_parent**"

A `VideoView` is a special kind of `SurfaceView` which includes a `MediaPlayer` to enable video display and playback. Let's add some functionality to it. Go to your `VideoPlayActivity` file and replace the provided `onCreate()` template with the following:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    val view = ActivityVideoPlayBinding.inflate(layoutInflater)
    setContentView(view.root)

    view.videoView.setVideoURI(intent.getParcelableExtra("VIDEO_URI"))

    with (MediaController(this)) {
        setAnchorView(view.videoView)
        view.videoView.setMediaController(this)
        view.videoView.setOnPreparedListener { show(0) }
    }
    view.videoView.setOnCompletionListener { finish() }
    view.videoView.start()
}
```

First we grab the `VIDEO_URI` passed to `VideoPlayActivity` from its caller (`MainActivity`). Then we set up a `MediaController`, a simple built-in controller, to be shown during video play back. We let `MediaController` be shown with timeout 0, otherwise, by default, it hides itself after 3 seconds. When the video stops playing, we close `VideoPlayActivity`.

`getChatts()`

Since we are using OkHttp3 to upload `chatt` in `postChatt()`, we could use OkHttp3 for download also. Replace your `getChatts()` method in `ChattStore` with:

```
fun getChatts(context: Context, completion: () -> Unit) {
    val request = Request.Builder()
        .url(serverUrl+"getimages/")
        .build()

    client.newCall(request).enqueue(object : Callback {
        override fun onFailure(call: Call, e: IOException) {
            Log.e("getChatts", "Failed GET request")
            completion()
```

```
            }

            override fun onResponse(call: Call, response: Response) {
                if (response.isSuccessful) {
                    val chattsReceived = try { JSONObject(response.body?.string() ?: "").getJSONArra

                    chatts.clear()
                    for (i in 0 until chattsReceived.length()) {
                        val chattEntry = chattsReceived[i] as JSONArray
                        if (chattEntry.length() == nFields) {
                            chatts.add(Chatt(username = chattEntry[0].toString(),
                                message = chattEntry[1].toString(),
                                timestamp = chattEntry[2].toString(),
                                imageUrl = chattEntry[3].toString(),
                                videoUrl = chattEntry[4].toString(),
                            ))
                        } else {
                            Log.e("getChatts", "Received unexpected number of fields " + chattEntry.
                        }
                    }
                    completion()
                }
            }
        })
    }
```

> If you prefer, instead of the OkHttp3-based `getChatts()` above, you could continue using the Volley-based `getChatts()` from lab0. Should you want to do that, find the `for` loop in your `getChatts()` and add code to extract the `imageUrl` and `videoUrl` strings (in order) from the retrieved chatt and pass them to the `Chatt` constructor, as we did above. If you're not using the Volley-based `getChatts()`, you can delete the `queue` property declaration in `ChattStore` and remove the volley dependency from your `build.gradle (Module)`.

We do not implement progressive image download and video streaming in this lab. So please be patient staring at a blank screen when downloading large images and/or playing back long videos, it can take a while for the video file to download fully before play back begins.

Congratulations, you've successfully added the ability to access your device's gallery or camera, upload/download images and videos to/from your server, and display images and play back videos in your app!

There is no special instructions to run lab1 on the android emulator.

Submission guidelines

IMPORTANT: If you work in team, remember to put your team mate's uniqnames in lab1 folder's `README.md` so that we'd know. Otherwise, we could mistakenly thought that you were cheating and accidentally report you to the Honor Council, which would be a hassle to all to undo.

Enter your uniqname (and that of your team mate's) and the link to your GitHub repo on the Lab Links sheet. The request for teaming information is redundant by design. If you're using a different GitHub repo from previous lab's, invite `eecs441staff@umich.edu` to your GitHub repo.

Push your lab1 to its GitHub repo as set up at the start of this spec. Using GitHub Desktop to do this, you can follow the steps below:

- Open GitHub Desktop and click on `Current Repository` on the top left of the interface
- Click on your `441` GitHub repo
- Add Summary to your changes and click `Commit to master`
- If you have a team mate and they have pushed changes to GitHub, you'll have to click `Pull Origin` and resolve any conflicts before . . .
- Finally click on `Push Origin` to push changes to GitHub

Go to GitHub website to confirm that your project files for lab1 have been uploaded to GitHub repo under folder `lab1` .

References

Android Camera

- Android's common camera app
- Android Camera Intent: how to get full sized photo?
- An Android Studio VideoView and MediaController Tutorial
- ImageView disappear after changed orientation
  - How to Lock Android App's Orientation to Portrait in Phones and Landscape in Tablets?
  - Android: Temporarily disable orientation changes in an Activity
  - How to set entire application in portrait mode only?
- Android: Let user pick image or video from Gallery
- Android Developer video basics

Activity Results

- Passing data from parent to child activites
- Passing data back from child to parent activities
- Activity Results API: A better way to pass data between Activities
- Deep Dive into Activity Results API – No More onActivityResult()

Misc. Android topics

- OnCompletionListener
- Understanding Activity.runOnUiThread()

- Extension functions in Kotlin: Extend the Android Framework (KAD 08)

Image download

- Introducing Coil: Kotlin-first image loading on Android
- Coil: Getting Started
- JPEG Formats - Progressive vs. Baseline
- Progressive JPEGs and green Martians

Image cropping

- Cropping saved images in Android
- Package visibility in Android 11
- No, Android Does *Not* Have a Crop Intent

Not updated to Android 11:

- Android cropping image from camera or galleryr
- Instagram like Image Cropper

Image upload

- HTTP/REST API File Uploads
- RESTful API Tutorial: How to Upload Files to a Server

OkHttp3

- okhttp3
- How does OkHttp get Json string?
- Posting a multipart request (.kt, .java)
- Android Http Requests in Kotlin with OkHttp
- Get Path From URI In Kotlin Android
- java.io.FileNotFoundException: /storage/emulator/0/New_file.txt: open failed: EACCES (Permission denied)
- How to convert `content://media/external/images/media/Y` to `file:///storage/sdcard0/Pictures/X.jpg` in android?
- Adding content to RequestBody
- Adding image as bytearray
- Kotlin - OkHttp - Return from onResponse

MediaStore and scoped storage

- How to save an image in Android Q using MediaStore

- How to save an image in a subdirectory on android Q whilst remaining backwards compatible
- How to save an image in Android Q using MediaStore?
- Scoped Storage in Android 10 & Android 11
- Storage Updates in Android 11
- The Quick Developers Guide to Migrate Their Apps to Android 11

Appendix: imports

Prepared for EECS 441 by Benjamin Brengman, Wendan Jiang, Alexander Wu, Ollie Elmgren, Tianyi Zhao, Yibo Pi, and Sugih Jamin	Last updated: July 4th, 2021