eecs441§3&4

Chatter Kotlin

Cover Page

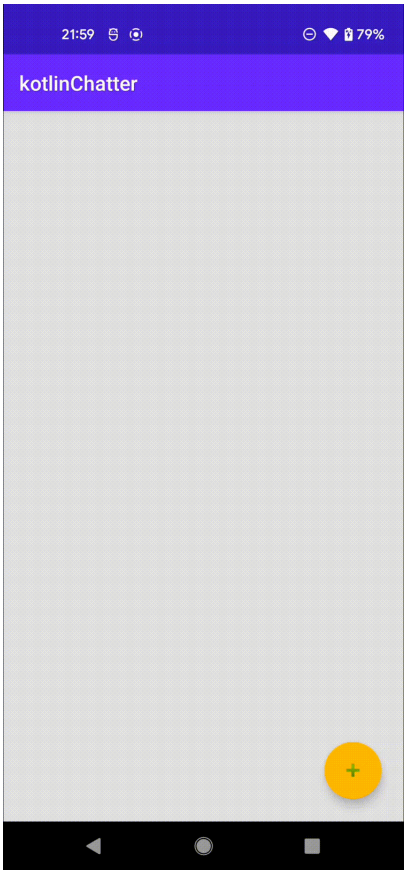
DUE Wed, 09/15, 2 pm

In this lab you'll learn how to retrieve textual `chatts` from a back-end server and how to post to it. You will familiarize yourselves with the Android app development environment and the basics of the platform. You'll learn some Kotlin syntax and language features that may be new to you. And you will be introduced to ConstraintLayout, Android's constraint-based layout mechanism, as used in the Layout Editor. Let's get started!

Gif demo

Posting a new chatt:

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



Preliminaries

If you don't have an environment set up for Android development, please read our note on Getting Started with Android Development first.

Before we start, you'll need to prepare a GitHub repo for you to submit your labs and for us to communicate your lab grades back to you. Please follow the instructions in Preparing GitHub for EECS 441 Labs and then return here to continue.

Creating an Android Studio project

In the following, please replace "YOUR_UNIQNAME" with your uniqname. Google will complain if your package name is not globally unique. Using your uniqname is one way to generate a unique package name.

Depending on your version of Android Studio, the screenshots in this and subsequent lab specs may not look exactly the same as what you see on screen.

1. Click `New Project` in the "Welcome to Android Studio" screen (screenshot)
2. On `Phone and Tablet` tab, select `Empty Activity` (should already be selected by default) and click `Next` (screenshot)

   ⚠ Choose `Empty Activity` **not** `No Activity`

3. Enter `Name` : kotlinChatter
4. `Package name` : edu.umich.YOUR_UNIQNAME.kotlinChatter

   > Android Studio may automatically change all upper case letters to lower case. If you prefer to use upper case, just edit the name again and it should take the second time.

5. `Save location` : specify the full path where your `kotlinChatter` folder is to be located, which will be `YOUR_LABSFOLDER/lab0/kotlinChatter/`

   > where `YOUR_LABSFOLDER` is the folder of your choosing where you want to put all your 441 labs.

6. `Language` : Kotlin
7. `Minimum SDK` : ANDROID_VERSION_OF_YOUR_PHONE

   ❗ The `Minimum SDK` must be **at least** `API 30: Android 11.0 (R)` . Android 12.0 (S, API Level 31) is also acceptable.

8. Click `Finish`

Subsequently in this and other labs, we will use the tag `YOUR_PACKAGENAME` to refer to your package name. Whenever you see the tag, please replace it with your package name.

Once the project is created, Android Studio will prompt you to `Add Files to Git` , hit `Cancel` . We will add the files to GitHub using GitHub Desktop instead.

> If you are proficient with git, you don't have to use GitHub Desktop. However, we can only help with GitHub Desktop, so if you use anything else, you'll be on your own.

All of your project files, including the gradle dependency management and build scripts, will be in `YOUR_LABSFOLDER/lab0/kotlinChatter`.

Android Studio project structure

We will assume that the left, navigation pane of your Android Studio window will show your project structure in `Android` view (screenshot) such that your project structure looks like this:

- `/app/manifests/AndroidManifest.xml` : general app settings and activity list
- `/app/java/PACKAGE_NAME` : source code
  > yes, it says `java` not `kotlin`
- `/app/res/drawable` : image assets
- `/app/res/layout` : UI View XML documents
- `/app/res/mipmap` : image assets at different resolutions
- `/app/res/values` : constants for strings and designs
- `/app/Gradle Scripts` : build scripts

> If your project pane doesn't look like the above, wait for Android Studio to finish syncing and building and configuring, your project should then be structured per the above

While we are not required to call the first activity of the app the `MainActivity` (it can be changed in `AndroidManifest.xml`), it is a convention to do so and Android Studio automatically sets up a new `Empty Activity` project assuming the first activity is so called.

Kotlin encourages closely-related declarations to be placed in the same source file, however, to reduce the temptation for creating MassiveViewControllers, in this course we will mostly follow Java's requirement and place each class in its own file.

Chatter

Consists of two views: one to write and post a `chatt` to server, and another, the main view, showing retrieved `chatts`. It is cheaply inspired by Twitter. And it has a live back-end API already:

```
https://mobapp.eecs.umich.edu/getchatts/
https://mobapp.eecs.umich.edu/postchatt/
```

We will create the back end in the second part of this lab.

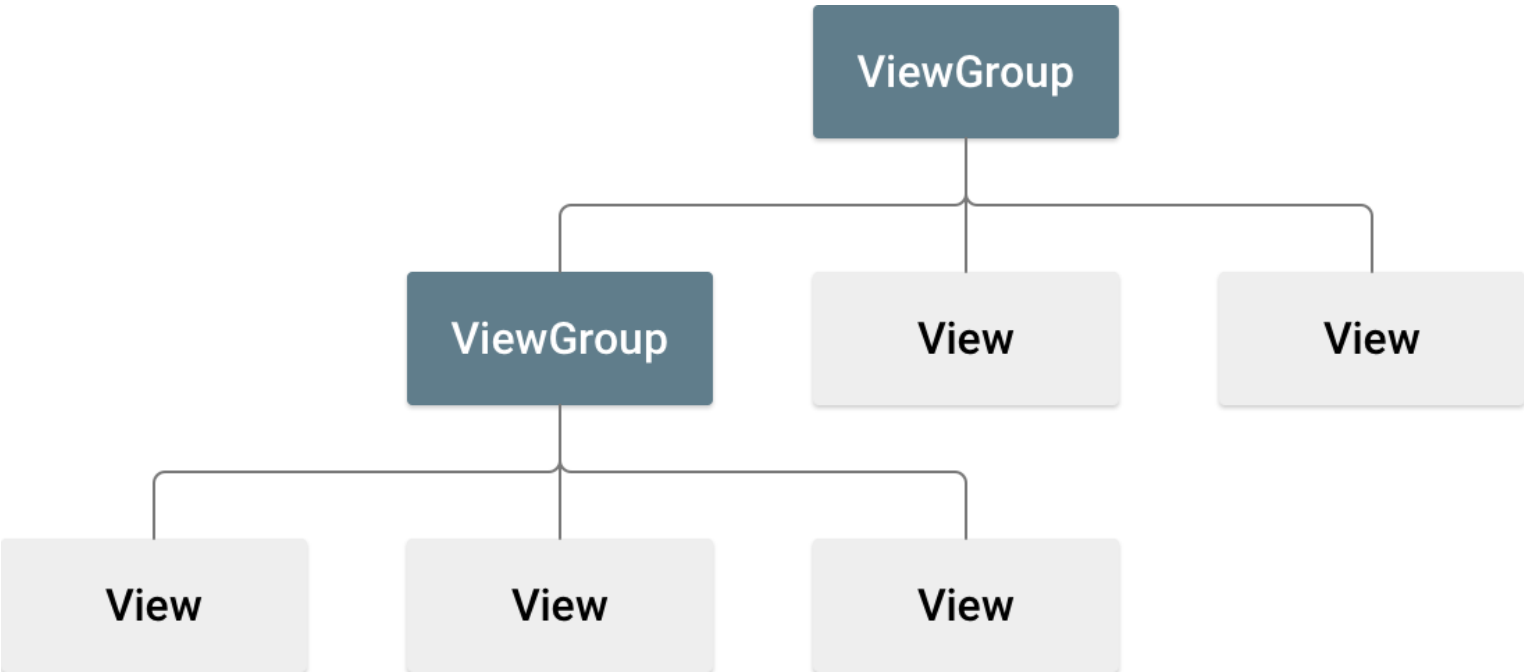
Layout Editor

For this lab, we will be using Android Studio's Layout Editor to construct our views. The Layout Editor helps us lay out our UI:

- Widgets (UI elements) are added to a screen layout by drag-and-drop.
- Click on the `Palette` pane in the Layout Editor to see the library of available widgets.

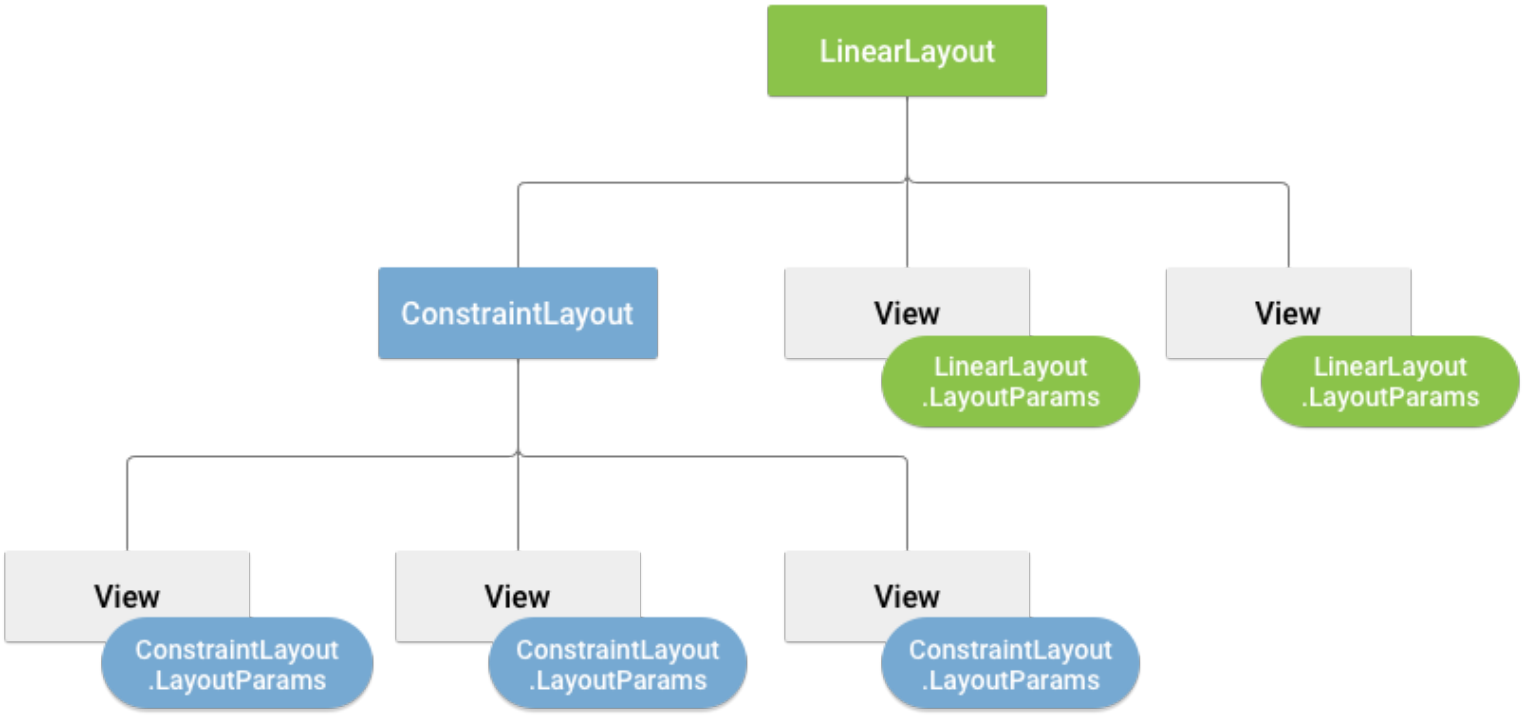
View and Widget , ViewGroup and Layout

A `View` is synonymous to a `widget` , i.e., a UI element such as a `Button` or a `TextView` . Similarly, for all intents and purposes, a `ViewGroup` is synonymous to a `Layout` , i.e., how UI elements are to be laid out on screen. A `ViewGroup` may be nested, thereby forming a view hierarchy. Here's a simple one from Google's documentation:



In this and subsequent labs, we will use `View` , `widget` , and `UI element` interchangably (though we won't be using `widget` much at all), and we will be using `ViewGroup` and `Layout` interchangably: `ViewGroup` when discussing grouping of `View` s and `Layout` when talking about how the `View` s are placed. To specify how UI elements are placed on the screen, Android originally supported `LinearLayout` , then `RelativeLayout` was introduced. Nowadays, we use `ConstraintLayout` .

Here is an example of a multi-layer ViewGroup. The root ViewGroup adopts LinearLayout: there are 2 UI elements and a second ViewGroup laid out linearly under this ViewGroup. The lower ViewGroup adopts ConstraintLayout and has 3 UI elements under it.



The above layout hierarchy could correspond to the following view, for example:

`ConstraintLayout`

With `ConstraintLayout`, how various UI elements (labels, buttons, tables, text fields, text boxes, images, etc.) are placed relative to each other, is automatically computed using a constraint satisfaction algorithm. `ConstraintLayout` sizes, resizes, positions, and repositions all UI elements as necessary based on available screen real estate and the constraints associated with each UI element. UI elements of an app are sized and positioned according to the screen size of the device running the app. UI elements are then resized and repositioned upon change of orientation, for example.

For `ConstraintLayout` to work, it needs to know four things about each UI element:

1. the x- and
2. y-coordinates of one of the element's **corner**s,
3. the element's **width**,
4. and its **height**.

Often though, we only need to specify one of the corners of a UI element and let its dimensions be automatically derived based on its content (`wrap_content`) or based on the constraints of its parent (`match_constraint` , achieved by setting width or height to `0dp`). There is also `match_parent` which is to be used outside `ConstraintLayout` , for example, when specifying the constraint of a `ConstraintLayout` itself

(remember, layouts can be nested, though nesting `ConstraintLayout` s within each other is not recommended).

The coordinate system for an Android screen has its origin (0,0) at the upper left corner of the screen (relative to orientation). X positive grows to the right and y positive grows down.

Source: Java Code Geeks

Different Android devices have different screen sizes (max x and y coordinates) (this table lists the screen sizes of a number of popular models as of March 2018). Here are the screen resolutions of the current Pixel phones (for a complete list, see Google's Pixel phone hardware tech specs):

Pixel 4	Pixel 4 XL	Pixel 4a	Pixel 4a 5G	Pixel 5
				
2280 x 1080 444 ppi	3040 x 1440 537 ppi	2340 x 1080 443 ppi	2340 x 1080 413 ppi	2340 x 1080 432 ppi

Source: New Atlas

To learn more about `ConstraintLayout`, I recommend reading this tutorial. Google's documentation, *Build a UI with Layout Editor* explains the Layout Editor's UI and *Build a Responsive UI with ConstraintLayout* shows how to use the Layout Editor to set constraints graphically. We also list additional articles in the references section below. Some of them have illustrations and animated gifs that may help clarify the concepts.

To help debug your layouts, you can give Google's Layout Inspector a try.

Let's design our screens!

First we define some strings we will be using in the app: open up `/app/res/values/strings.xml`, inside the `resources` block, below the line listing your `app_name`, add:

```
<string name="post">Post</string>
<string name="send">Send</string>
<string name="username">YOUR_UNIQNAME</string>
<string name="message">Some short sample text.</string>
```

> Replace YOUR_UNIQNAME with your uniqname.

Main view

1. Open `/app/res/layout/activity_main.xml`.
2. Click on the `Design` icon to start the Layout Editor, if it's not already running (see figure below).
3. Click on the blue `Layer` icon and select `Blueprint` only (no `Design`).
4. Delete the default `TextView` (containing `Hello World!`) (screenshot).

Add a `FloatingActionButton` to compose and post `chatt`

1. On the `Palette` pane on the left side of the editor, navigate to `Buttons > FloatingActionButton` and click on the download icon. A dialog box will pop up saying that you need to Add Project Dependency and download a google library. Click `OK`.
2. Drag and drop a `FloatingActionButton` to the lower right corner of your design blueprint.

3. Select a `Drawable > ic_input_add` icon, click `OK`.
4. Click on the `Attributes` drawer (screenshot).
5. Change the `FloatingActionButton`'s `id` to "postButton" (screenshot)
6. Scroll down until you see the `Layout > Constraint Widget` section; click on the bottom and right white-on-blue plus signs to set the bottom and right constraints (16 dp suggested) (screenshot).
7. You can additionally change the `backgroundTint` of the `FloatingActionButton` to `#FFC107`.

> You can experiment with other colors by consulting the Material Design Color System (scroll all the way down until you get to the "2014 Material Design color palettes").

▶ What's "dp"?

Add a `ListView` to show retrieved `chatt s`

1. From the `Palette` pane, drag and drop a `Legacy > ListView` onto your design blueprint (put it anywhere on the blueprint).
2. In the `Declared Attributes` pane, give it `id` "chattListView".
3. Set its `layout_width` and `layout_height` to `0dp (match_constraint)` in the drop down menu.

We'll work on Post view next and return to populating the `ListView` afterwards.

Post view

To work on the Post view, we create a new `PostActivity` and let Android Studio create a view template for us:

1. Right click on `/app/java/` and select `New > Activity > Empty Activity`.
2. Call the new activity `PostActivity`, leave the rest of the form as per default, click `Finish` and on the pop up dialog box click `Add` (you may want to check `Remember, don't ask again` also).

Now open up `/app/res/layout/activity_post.xml`. Confirm that you're in `Design` mode. Then:

Add `usernameTextView`

1. From the `Palette` pane, drag and drop a `Common > TextView` onto your blueprint to hold the username.
2. In the `Declared Attributes` section on the right pane, set the `TextView id` to "usernameTextView" and enter "@string/username" in the `text` field. Set `textSize` to `24sp`.

▶ What's "sp"?

1. In the `Layout > Constraint Widget` section; click on the top white-on-blue plus sign and set it to 22dp. If you look back up to the `Declared Attributes` section, you should see `layout_constraintTop_toTopOf` set to `parent` and `layout_marginTop` set to 22dp.

> The parent of a UI element is the container in which the UI element resides. In the `Component Tree` pane of the `Design` mode, the parent of a UI element is the element one level higher up in the tree. In this case, the parent of `usernameTextView` is `ConstraintLayout`, which refers to the layout container of the screen.

1. Back in the `Constraint Widget` section, click on both the left and right white-on-blue plus signs and set both margin constraints to 0dp of parent. This should position the `TextView` centered horizontally. Looking back up to the `Declared Attributes` section, you should now see `layout_constraintStart_toStartOf` and `layout_constraintEnd_toEndOf` both set to `parent`.

> ℹ If your language localization is for a language that reads left to right, `Start` is the same as `Left`, otherwise for languages read right to left (RTL), `Start` is the same as `Right`. Conversely `End`. Most of the time you would use `Start`, `End`, reserving `Left` and `Right` only when you need to explicitly refer to physical-world left or right, e.g., when giving direction.

Add `messageTextView`

1. Add a `Text > Multiline Text` for the message.
2. In the pane showing an image of the screen, drag the circle at the top of the `messageTextView` rectangle and point it to circle at the bottom of the box with your uniqname.
3. In the `Layout > Constraint Widget` section set the top margin to 20. Click on the Start/Left plus sign and set its margin to 0.
4. Give it `id` "messageTextView" and confirm that its `width` and `height` are set to `wrap_content`.
5. Scroll down the `Attributes` pane to the `All Attributes` section, find the attribute `text` and enter "@string/message". Set `textSize` to `18sp`. (The attributes are listed alphabetically.)

Your Post view should look something like this screenshot.

1. If Android Studio complains that the `autofillHints` is not set, click `Fix` next to `Set importantForAutofill="no"`.

`ConstraintLayout`

On the `Component Tree` pane, click on `ConstraintLayout`, then in the `Attributes` pane, scroll down to `padding`, open up the section by clicking on the `>` and enter `16dp` on the first entry (also called `padding`) (screenshot).

> ▶ Margins vs. padding

Chatt list item layout

1. Select the `/res/layout/` directory
2. Right click `New > Layout Resource File`

3. Name file `listitem_chatt`, leave the rest of the fields as per default and click `OK`

With the newly created `listitem_chatt.xml` loaded on the Layout Editor (`Design` mode):

1. Add a `Common > TextView` for the `username`, constrained to Top and Start/Left. In its `Attributes` pane, set its `id` to "usernameTextView", `text` to "@string/username", and `textSize` to `18sp`. Confirm that both its `layout_width` and `layout_height` are set to `wrap_content`.

> You can safely ignore Android Studio's warning that we're re-using the same id.

1. Add another `Common > TextView` for the `timestamp`, constrained it to Top and End/Right. Set its `id` to "timestampTextView" and its `textSize` to `14sp`. Confirm that both its `layout_width` and `layout_height` are set to `wrap_content`.

2. Add a `Common > TextView` for the message, set its `id` to "messageTextView", `text` to "@string/message", and `textSize` to `18sp`.

> Again, you can safely ignore Android Studio's warning that we're re-using the same id.

1. Make `messageTextView` full-width:
   i. constrain both left and right margins to the default 0dp,
   ii. set `layout_width` to `0dp (match_constraint)`
   iii. constrain its top to `8dp` off the bottom of `usernameTextView`

`ConstraintLayout`

On the `Component Tree`, select the `ConstraintLayout` and set its padding to:

- paddingStart: 6dp
- paddingTop: 8dp
- paddingEnd: 6dp
- paddingBottom: 14dp

> Earlier, we said that it's not recommended to nest a ContraintLayout within another. I suppose a `ListView` item layout would be an exception, due to how it is reused and controlled by an Adapter.

We won't be grading you on how beautiful your UI looks, all the constraints suggested in this and all subsequent labs are suggestions. You're free to design your UI differently, so long as all indicated UI elements are visible and functional.

Navigation between activities

In `MainActivity.kt` add the following method to the `MainActivity` class:

```
fun startPost(view: View?) = startActivity(Intent(this, PostActivity::class.java))
```

In `/res/layout/activity_main.xml` set the `postButton`'s `onClick` attribute to "startPost" (screenshot).

In `/manifest/AndroidManifest.xml` find the line with `.PostActivity` and replace it with:

```
<activity android:name=".PostActivity"
    android:label="@string/post"
    android:parentActivityName=".MainActivity" />
```

This adds a back arrow in the `ActionBar` at the top of `PostActivity` view and gives it the title `Post` .

Posting and retrieving `chatts`

We are done with the UI layout and navigation. Let's move on to code for posting and retrieving `chatts` .

Permission and dependency

First we need user's permission to use the network. In `AndroidManifest.xml` , before the `<application` block, add:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

In file `Gradle Scripts/build.gradle (Project:kotlinChatter.app)` , in the `buildscript` block add:

```
ext {
    kotlin_version = '1.5.30'
}
```

In file `Gradle Scripts/build.gradle (Module:kotlinChatter.app)` (note this is the **Module** gradle file **not** the **Project** gradle file above), in the `android` block add:

```
buildFeatures {
    viewBinding true
}
```

`ViewBinding` allows us to refer to UI elements in layout files by their `android:id` tags.

Further down, in the `dependencies` block, add:

```
implementation 'com.android.volley:volley:1.1.0'
implementation 'androidx.swiperefreshlayout:swiperefreshlayout:1.0.0'
implementation "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"
```

`Volley` is a native android library used to make HTTP requests. We will use `SwipeRefreshLayout` to initiate retrieval of new `chatt` s.

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` , click `OK` .

`Chatt` class

To post a `chatt` with the `postchatt` API, `Chatter` back end server expects a JSON object consisting of "username" and "message". For example:

```
{
   "username": "YOUR_UNIQNAME",
   "message": "Hello world!"
}
```

`Chatter`'s `getchatts` API will send back all accumulated chatts in the form of a JSON object with the key being "chatts" and the value being an array of string arrays. Each string array consists of three elements "username", "message", and "timestamp". For example:

```
{
    "chatts": [["username0", "message0", "timestamp0"],
               ["username1", "message1", "timestamp1"],
               ...
              ]
}
```

Each element of the string array may have a value of JSON `null` or the empty string (`""`).

Create a new Kotlin file:

1. Select `/app/java/PACKAGE_NAME` directory
2. Right click: `New > Kotlin Class/File`
3. Name `Chatt` and double click on `File` (screenshot)
4. Place the following class definition for `Chatt` in the newly created file:

```
class Chatt(var username: String? = null,
            var message: String? = null,
            var timestamp: String? = null)
```

`ChattStore` as Model

We will declare a `ChattStore` object to hold our array of `chatt` s. `ChattStore` will serve as the Model of our app, following the Model-View-Controller architecture. Since the `chatt` s are retrieved from the `Chatter` back-end server and sent to the same back-end server when the user posted a `chatt` , we will keep the network functions to communicate with the server as methods of this class also.

Create another Kotlin file, call it `ChattStore` , and place the following `ChattStore` object in it:

```
object ChattStore {
    val chatts = arrayListOf<Chatt?>()
    private val nFields = Chatt::class.declaredMemberProperties.size
```

```
    private lateinit var queue: RequestQueue
    private const val serverUrl = "https://mobapp.eecs.umich.edu/"
}
```

Using the keyword `object` to declare the class makes it a **singleton** object, meaning that there will ever only be one instance of this class when the app runs. Since we want only a single copy of the `chatt` s data, we make this a singleton object.

The `chatts` array will be used to hold the `chatt` s retrieved from the back-end server. The code `Chatt::class.declaredMemberProperties.size` uses introspection to look up the number of properties in the `Chatt` type. We store the result in the variable `nFields` for later validation use.

> Once you have your own back-end server set up, you will replace `mobapp.eecs.umich.edu` with your server's IP address.

Posting `chatt`

Add the following method to your `ChattStore` object above:

```
    fun postChatt(context: Context, chatt: Chatt) {
        val jsonObj = mapOf(
            "username" to chatt.username,
            "message" to chatt.message
        )
        val postRequest = JsonObjectRequest(Request.Method.POST,
            serverUrl+"postchatt/", JSONObject(jsonObj),
            { Log.d("postChatt", "chatt posted!") },
            { error -> Log.e("postChatt", error.localizedMessage ?: "JsonObjectRequest error") }
        )

        if (!this::queue.isInitialized) {
            queue = newRequestQueue(context)
        }
        queue.add(postRequest)
    }
```

We first assemble together a Kotlin map comprising the key-value pairs of data we want to post to the server. To post it, we create a `JsonObjectRequest()` with the appropriate POST URL. We can't just post the Kotlin map as is though. The server may not, and actually is not, written in Kotlin, and very likely will have a different memory layout for various data structures. Presented with a chunk of binary data, the server will not know that the data represents a map, nor how to reconstruct the map in its own map layout. To post the Kotlin map, therefore, we call `JSONObject()` to encode the Kotlin map into a serialized JSON object that the server will be able to parse.

Once the POST request is created, we submit it to the request queue managed by the Volley networking library for asynchronous execution. Prior to submitting the request to the request queue, we check that the queue has been created and, if not, create the queue. We will discuss the use of `Context` further in the call to `postChatt()` below.

▶ Kotlin serialization

PostActivity

Now, we turn to your `PostActivity` class. Replace the `onCreate()` method of your `PostActivity` class with the following:

```
private lateinit var view: ActivityPostBinding
private var enableSend = true

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    view = ActivityPostBinding.inflate(layoutInflater)
    setContentView(view.root)
}
```

When Android Studio creates a new project for us, it automatically assigns a default `AppTheme` to the app (`android:theme:"@style/AppTheme"` in `AndroidManifest.xml`). Among other things, the default theme specifies a default `ActionBar` atop each of our Activity screen. The `ActionBar` is where the Activity title, navigation icons (e.g.,back arrow) and the `Options Menu` can be found (screenshot). (See references below to learn more about Android's themes, action bar, and options menu.)

We now add a `Send` item to the `Options Menu` on the `ActionBar` of our `PostActivity` screen. Unless specified otherwise, menu items of `Options Menu` are listed on a drop-down menu under an `Overflow Button` (three vertical dots) (screenshot). We can however, depending on screen size, have up to two of the menu items comfortably shown on the `ActionBar` itself as icons, which we will do next.

Now add the following two methods to your `PostActivity` class to set up the `Send` icon on the `ActionBar` , which will post the `chatt` when tapped.

```
override fun onPrepareOptionsMenu(menu: Menu?): Boolean {
    menu?.apply {
        add(NONE, FIRST, NONE, getString(R.string.send))
        getItem(0).setIcon(android.R.drawable.ic_menu_send).setEnabled(enableSend)
                .setShowAsAction(MenuItem.SHOW_AS_ACTION_ALWAYS)
    }
    return super.onPrepareOptionsMenu(menu)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    if (item.itemId == FIRST) {
        enableSend = false
        invalidateOptionsMenu()
        submitChatt()
    }
    return super.onOptionsItemSelected(item)
}
```

The `Menu` method `setShowAsAction(MenuItem.SHOW_AS_ACTION_ALWAYS)` tells Android to show this `Options Menu` item as an icon on the `ActionBar` instead of a list item on a drop-down menu. Once user has clicked the `Send` button, we set `enableSend` to `false` and call `invalidateOptionsMenu()`, which causes Android to run `onPrepareOptionsMenu()` again. With `enableSend` set to `false`, the `Send` button will be disabled and "greyed out" when rendered by `onPrepareOptionsMenu()`. This feature is not as visible in this lab since the sending process completes immediately. In later labs, when the sending process can take some time, it prevents user from repeatedly clicking the `Send` button.

Finally add the following `submitChatt()` function to the `PostActivity` class:

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

    postChatt(applicationContext, chatt)
    finish()
}
```

Recall that we need only one request queue in `ChattStore`, but `ChattStore`, being an `object`, persists for the whole lifetime of the app. The `Context` off which we create the request queue thus also needs to persist for the whole lifetime of the app, which is why we pass the `applicationContext` to `postChatt()` here, as opposed to passing the activity's context (`this`), which is destroyed when we dismiss `PostActivity` or on orientation change.

Retrieving `chatt` s

Setting up the recyclable list adapter `ChattListAdapter`

Recall that an `ArrayAdapter` is used to link items in a list to its view. It is the controller that intermediates between the view and the model. Create a new Kotlin file, `ChattListAdapter` and put the following class `ChattListAdapter` in the file:

```
class ChattListAdapter(context: Context, users: ArrayList<Chatt?>) :
    ArrayAdapter<Chatt?>(context, 0, users) {

    override fun getView(position: Int, convertView: View?, parent: ViewGroup): View {
        val listItemView = (convertView?.tag /* reuse binding */ ?: run {
            val rowView = LayoutInflater.from(context).inflate(R.layout.listitem_chatt, parent, fals
            rowView.tag = ListitemChattBinding.bind(rowView) // cache binding
            rowView.tag
        }) as ListitemChattBinding

        getItem(position)?.run {
            listItemView.usernameTextView.text = username
            listItemView.messageTextView.text = message
            listItemView.timestampTextView.text = timestamp
            listItemView.root.setBackgroundColor(Color.parseColor(if (position % 2 == 0) "#E0E0E0" e
        }

        return listItemView.root
```

```
    }
}
```

The `getView()` method of the adapter first checks if a recycled `View` has been passed in. If so, it re-uses the view binding for that view that is saved in the view's `tag` field to re-populate the recycled view. If not, it creates a new `View`, inflates (creates UI elements for) it according to the layout stored in `R.layout.listitem_chatt`, and binds the variables in `ListItemChattBinding` to the created UI elements. This binding is then stored in the view's `tag` field so that we will have ready access to it when the view is recycled, without having to re-do the bindings.

Once all the UI elements of the list item are populated, we set the background color to alternate between light grey and very-light grey.

MainActivity

To retrieve `chatt`s on app launch, replace `onCreate()` in `MainActivity` with:

```
private lateinit var view: ActivityMainBinding
private lateinit var chattListAdapter: ChattListAdapter

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    view = ActivityMainBinding.inflate(layoutInflater)
    view.root.setBackgroundColor(Color.parseColor("#E0E0E0"))
    setContentView(view.root)

    chattListAdapter = ChattListAdapter(this, chatts)
    view.chattListView.setAdapter(chattListAdapter)

    // setup refreshContainer here later
}
```

We set the background color here so that the `ListView` doesn't show up with white margin around it. Following the Model-View-Controller architecture, the `ListView` is the view displaying lists, the array carrying the content to be displayed is the model, the `ArrayAdapter` is the controller that intermediates between the view and the model. Here we construct a `ChattListAdapter`, which is a subclass of `ArrayAdapter`, with an empty `ArrayList()` to hold the model (i.e., the `chatt`s to be retrieved from the back end server), then we associate the `chattListView` with the adapter.

Congratulations! You have set up a `ListView` with its `ArrayAdapter` and bind both to the `chatts` array. Next we need to retrieve the `chatt`s from the `Chatter` back end and show them on the timeline in the main screen.

getChatts()

Add the following `getChatts()` method to the `ChattStore` object in `ChattStore.kt`:

```
fun getChatts(context: Context, completion: () -> Unit) {
    val getRequest = JsonObjectRequest(serverUrl+"getchatts/",
```

```
        { response ->
            chatts.clear()
            val chattsReceived = try { response.getJSONArray("chatts") } catch (e: JSONException
            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()))
                } else {
                    Log.e("getChatts", "Received unexpected number of fields: " + chattEntry.len
                }
            }
            completion()
        }, { completion() }
    )

    if (!this::queue.isInitialized) {
        queue = newRequestQueue(context)
    }
    queue.add(getRequest)
}
```

To retrieve `chatt` s, we create a `JsonObjectRequest()` with the appropriate GET URL. The server will return the `chatt` s as a JSON object. In the completion handler to be invoked when the response returns, we call `.getJSONArray()` to decode the serialized JSON value corresponding to the `"chatts"` key from the return response.

Pull-down to refresh

The list of retrieved `chatts` is not automatically refreshed. We implement a pull-down to refresh feature instead.

Navigate to `/app/res/layout/activity_main.xml` , set editor to `Code` mode), and embed your `ListView` within a `SwipeRefreshLayout` . Replace your `ListView` block with:

```
<androidx.swiperefreshlayout.widget.SwipeRefreshLayout
    android:id="@+id/refreshContainer"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent">

    <ListView
        android:id="@+id/chattListView"
        android:layout_width="0dp"
        android:layout_height="0dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.swiperefreshlayout.widget.SwipeRefreshLayout>
```

Then in `MainActivity`, add the following `refreshTimeline()` method:

```
private fun refreshTimeline() {
    getChatts(applicationContext) {
        runOnUiThread {
            // inform the list adapter that data set has changed
            // so that it can redraw the screen.
            chattListAdapter.notifyDataSetChanged()
        }
        // stop the refreshing animation upon completion:
        view.refreshContainer.isRefreshing = false
    }
}
```

We again pass `applicationContext`, instead of the activity's context (`this`), to `getChatts()` because the activity can be destroyed on orientation change, for example. In the completion handler, we notify the `chattListAdapter` of any data changes so that it can update the screen, which can only be done on the main/UI thread.

To retrieve `chatts` on app launch, and to set up the pull-down to refresh controller, add the following code **inside** `onCreate()` of `MainActivity`, where we left the comment, `// setup refreshContainer here later`:

```
    // setup refreshContainer here later
    view.refreshContainer.setOnRefreshListener {
        refreshTimeline()
    }

    refreshTimeline()
```

Congratulations! You're done with the first lab!

Run and test to verify and debug

If you're not familiar with how to run and test your code, please review the instructions in the Getting Started with Android Development.

There is no special instructions to run lab0 on the android emulator.

Submission guidelines

> ❗ **IMPORTANT:** If you work in team, put your team mate's name and uniqname in your repo's `Readme.md` (click the pencil icon at the upper right corner of the `Readme.md` box on your git repo) 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 undo.

Invite `eecs441staff@umich.edu` to your GitHub repo. 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.

Push your lab0 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 GitHub repo you created above, at the very start of this lab
- Add Summary to your changes and click `Commit to master` at the bottom of the left pane
- If you have a team mate and they have pushed changes to GitHub, you'll have to click `Pull Origin` and resolve any conflicts, re-commit to master, and
- Finally click on `Push Origin` to push changes to GitHub

Go to the GitHub website to confirm that your project files for lab0 have been uploaded to your GitHub repo under folder `lab0`.

Miscellaneous

Possible improvements

- Error handling
- Carefully spaced layouts
- Ability to cancel posting
- Refresh only new `chatts`, don't load everything again
- Efficient asynchronous requests to APIs

References

General

- Android Developers web site
  - Material Design for Android
  - Android Studio
  - Android Kotlin Training
  - Kotlin reference and Android tutorials
  - Android app activity lifecycle
  - Context, What Context?
  - Context and memory leaks in Android
  - How to Simplify Networking In Android: Introducing The Volley HTTP Library
  - Volley tutorial
  - Check whether a `lateinit var` is initialized

    - Use view binding to replace findViewById
- Gradle
- Google Play developer account
- GooglePlay's Testing Tracks
- Publishing app to Play Store

Styles, Themes, ActionBar, Menus

- Styles and Themes
- Material Design Color System scroll all the way down until you get to the "2014 Material Design color palettes"
- Change FloatingActionButton backgroundTint
- Menus
- Options Menu in Android
    - How to enable/disable option menu item on button click?
- Material Design Icons
- Material Design Icons Guide
- Add multi-density vector graphics
    - updated

Layout, screen density, margins

- Layouts
- Debug Your Layout with Layout Inspector and Layout Validation
- Designing for multiple screen densities on Android
- Difference Between dp, dip, sp, px, in, mm, pt in Android
- Screen sizes and densities market distribution
- Screen compatibility overview

Constraint Layout

- Android Constraint Layout
- Build a UI with Layout Editor
- Build a Responsive UI with ConstraintLayout
- ConstraintLayout in the LIMELIGHT
    - ConstraintLayout: NEVER EVER!
- Building interfaces with ConstraintLayout with animated gifs
- Android Constraint Layout using Kotlin more animated gifs
- ConstraintLayout yet more illustrations and animation
- Advanced ConstraintLayout use of constraint-satisfaction algorithm
- Understanding the performance benefits of ConstraintLayout

Appendix: imports

Prepared for EECS 441 by Tiberiu Vilcu, Yibo Pi, and Sugih Jamin	Last updated: June 30th, 2021