Chatter with Audio and Jetpack Compose Kotlin

Cover Page

DUE Wed, 10/27, 2 pm

We have two goals with this lab: first, to introduce you to declarative UI development using Android Jetpack Compose, with its reactive programming framework, and second, to add audio support to `Chatter`. Let's get started!

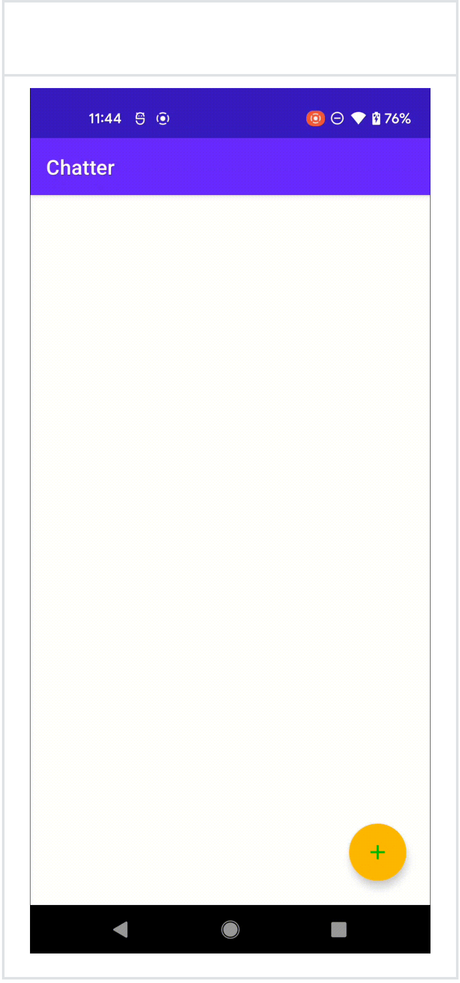
> ⚠️ This lab requires Android 12. If you have a Pixel 3 or later, you can upgrade to Android 12; otherwise, you would have to use the Android emulator to complete this lab. Please refer to Getting started with Android development if you need help checking the version of Android running on your phone or to set up an Android 12 virtual device on the emulator.
>
> A reminder that Jetpack Compose requires Android Studio Arctic Fox 2020.3.1 or later. You can check your Android Studio version in Android Studio:
>
> - on macOS: `Android Studio > About Android Studio`, or
> - `Help > About` otherwise.

Gif demo

Post chatts with/without audio:

Note: Annotations in orange describe user actions or screens not recorded by the screen recorder and are not part of the app.

> 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).



Part I: Converting Lab0 to Jetpack Compose

We will program the UI and UX flow of Chatter in Android Jetpack Compose. Chatter UI's appearance and behavior will change as reaction to changes in the underlying Audio Player's state, following a state machine. Except for the Model part of Chatter (the `Chatt` and `ChattStore` objects), to adopt Jetpack Compose means the rest of the app must be rewritten: to describe the UI declaratively and to set up the observer-pattern "plumbing" so that the UI can react to underlying state changes.

We have prepared a *Why Declarative-Reactive UI?* write up that lists the pros and cons of building UI using the declarative-reactive approach. Please give it a read if you're interested.

Creating an Android Studio project for Jetpack Compose

- On your laptop, navigate to `YOUR_LABSFOLDER/`
- Create a new folder and name it **lab3**
- Push your local `YOUR_LABSFOLDER/` repo to GitHub and make sure there're no git issues

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.

1. Launch Android Studio and create a new project.
2. On `Phone and Tablet` tab, select `Empty Compose Activity` and click `Next` (screenshot)
3. Enter `Name`: kotlinJpCChatter
4. `Package name`: edu.umich.YOUR_UNIQNAME.kotlinJpCChatter

> replace `YOUR_UNIQNAME` with yours
>
> 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 your `kotlinJpCChatter` folder is to be located, which will be `YOUR_LABSFOLDER/lab3/kotlinJpCChatter/`
6. `Language` : Kotlin
7. `Minimum SDK` : API 31: Android 12.0 (S)

> ❗ The `Minimum SDK` must be **at least** `API 31: Android 12.0 (S)` .
>
> Do NOT click `Use legacy android support libraries` .

8. Click `Finish`

As with lab0, when Android Studio prompts you to `Add Files to Git` , hit `Cancel` . We will add the files to GitHub using GitHub Desktop instead.

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 (you can leave `:app >> junit:junit:4.+` alone), click `Apply` , click `OK` .

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). Compared to the navigation pane in previous labs, you should see the following new items:

- `/app/java/PACKAGE_NAME/` :
  - ui.theme/: the files here specify the “theme” (color, shape, typeface, etc.) of your app, it defaults to the Material Design theme.
    - Color.kt: ARGB definition of color, see also /app/res/values/colors.xml
    - Shape.kt: definition of shape, rounded corner
    - Theme.kt: Material Theme with dark and light modes
    - Type.kt: definition of type faces, weights, and sizes
  - MainActivity.kt: we will be modifying this source file later.

▶ Theming

- /app/res/:
  - values:
    - themes: color definitions for theme (cannot delete)

> ⚠ While Jetpack Compose is highly anticipated by the Android developer community and has become more mature since its introduction in 2019, it just turned 1.0-stable in July 2021. Compared to

conventional Android View, it still lacks common features such as: swipe down to refresh and has a rather thin and basic community support.

And Android Studio does not seem to be so all knowing of Jetpack Compose APIs as to always make the relevant completion suggestions or bring up the relevant documentations.

Jetpack Compose and MutableState

We will look at several concepts that define an app built using Jetpack Compose and MutableState:

1. Declarative UI
2. Nested linear layouts
3. Immutability of Composed View and Recomposition
4. Separation of Composed View and State
5. Single-Authority State

If you are familiar with the Model-View-Controller sofware architecture and the Constrained Layout used with conventional Android Views, it may be easiest to think of Jetpack Compose as having the Controller and ConstrainedLayout functionalities prefabricated and baked-in to the Jetpack Compose runtime, so that we only need to program the View and the Model.

Declarative UI

A `Compose` or `@Composable` is a function that describes what UI elements a View contains: whether it has text boxes, images, labels, buttons, etc., and how these elements are laid out. So a `Compose` replaces XML files of conventional Android Views or the `ConstraintLayout` class of its programmatic equivalent. In Jetpack Compose, a `Compose` is thus the equivalent of a `View`, a `ViewGroup`, and a `Layout`. Subsequently, we will consider the terms `Compose`, `Compose function`, `@Composable`, `@Composable function`, `Composable function` as equivalent and will simply refer to the lot as `composable`. In Jetpack Compose, we write composable to create UI, $UI = f(State)$. A composable takes data as input and emits a view hierarchy as output.

A composable can also be used to customize individual UI element with `Modifiers`: what size and color font, whether text lines wrap around or got cut off, whether boxes have rounded corners, what color foreground and background, etc., though if these “design” features are to be uniform across the whole app, it is best to describe them in the `theme` files.

Nested linear layouts

Compose handles nested layouts efficiently. Developers no longer has to rely on constraints to position UI elements in a flat layout so as to avoid nested linear layouts for performance reasons, as in Android Views. Instead, one can use the simpler linear layout where the developer has a choice of layout directions: in a `Row` (along the x-axis, left to right), in a `Column` (y-axis, top to bottom), or as `Box`, which allows stacking items on top of each other (along the z-axis, back to front). As UI elements are added, they are automatically kept aligned with evenly distributed spacing. There are also lazy versions of these: `LazyRow` and `LazyColumn`, which do not create and load the whole view at once—useful for viewing large data set. (There's also an experimental `Grid` layout.)

There are also `alignment` modifiers one can specify when invoking a layout stack: whether elements should be aligned along their top edges, along their start edges, etc. `Arrangement` modifiers specify whether and by how much the first and last elements are offset from the edges of the view.

> ℹ Building UI with Jetpack Compose is building declaratively: one describes the UI without laying it out manually. Instead, the runtime systems applies some prefabricated layout rules on the UI elements. If one is happy with the default layouts (e.g., even spacing between UI elements), building UI with Jetpack Compose is a lot less complicated than building with Layout Editor, XML, or programmatically.
>
> ▶ constraintlayout-compose

Immutability of Composed View and Recomposition

Once composed, Views are not changed. Modifiers that decorate or add behavior to view composition are **ordered** and **immutable**. The ordering of modifiers matters: applying a background color and then padding will have the padding hiding the background color; whereas applying padding first and then a background color will see the padding painted with the background color.

When the data shown in a View changes, the View is not modified. Instead, the View is **recomposed**: a whole new View is constructed from scratch and re-rendered. And this is done automatically by virtue of reactive programming.

Separation of View and State

A composed View does not hold any **application data** (though it can hold view-logic data). There is no need to manually move data between View and Model. Instead, we rely on **data binding** and the **Observer Pattern** for data updates to be propagated between Model and View by the runtime systems.

When a piece of data changes, all the Views (`RecomposeScope` s) that subscribe to it will "react" to the change by recreating and re-rendering each View according to the new data: $UI = f(State)$

Single-Authority State

Variously known as "single source of truth," this means there is only one authoritative copy (**Subject**) of app data. All "copies" of a piece of data are either references to the same data (two-way bindings) or immutable copies of the data. By virtue of reactive programming, all references to, or **Observer**s of, a piece of data will be notified by the runtime systems if the data changes: $State' = f(State)$

For further explanation on state management, see explanations below of `mutableStateOf()` and `mutableStateListOf()`. See the explanation for `remember()` and `rememberSaveable()` for a discussion on interaction between View recomposition and State.

The **Observer Pattern** is closely related to the **Publisher-Subscriber Pattern**. In the **Observer Pattern**, a piece of data/state, called the **Subject**, is “observed” by one or more **Observers**. When the state changes, the observers are notified. Whereas the original MVC stipulated the use of the Observer Pattern, the propagation of change must be set up by the developer manually/imperatively. With declarative UI, the developer simply “declares” which variables are involved in an observer relationship and the propagation of change is set up and delivered automatically by the reactive framework.

In the **Publisher-Subscriber Pattern**, the goal is the same, but the implementation is more decentralized. When a “published” state changes, the change is “announced” in a broadcast medium, e.g., a bulletin board, an event bus, a multicast group, etc., that the subscribers listen on. So the difference is that in the “pub-sub” pattern, the subscribers are not individually notified, but rather learn about the change by group messaging. (See also Observer vs Pub-Sub pattern.)

In our labs, since we’re not concerning ourselves with the system’s implementation of the reactive framework, we will not be making this distinction between Observer and Pub-Sub patterns. For example, it is natural to say that the “Subject” is “published” and that the “Observers” are “subscribed to published state.” Consequently, we will use “Subject” and “Publisher” interchangably and similarly use “Observers” and “Subscribers” interchangably.

Chatter

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

```
    <string name="chatter">Chatter</string>
    <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.

Permission, navigation, 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 (Module:kotlinJpCChatter.app)` , in the `dependencies` block, add:

```
    implementation 'com.android.volley:volley:1.2.1'
    implementation 'org.jetbrains.kotlin:kotlin-reflect:1.5.21'
    // https://google.github.io/accompanist/swiperefresh/ doesn't always auto update
    implementation 'com.google.accompanist:accompanist-swiperefresh:0.19.0'
    implementation 'androidx.navigation:navigation-compose:2.4.0-alpha06' // 07 on require minSdk 31
```

`Volley` is a native android library used to make HTTP requests. We will use `SwipeRefresh` from the third-party "Accompanist" library to initiate retrieval of new `chatt` s. `Navigation compose` helps us navigate between views and `Kotlin reflect` helps with code introspection, as will be explained when used in `ChattStore` below.

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` .

As mentioned ealier, the Model part of the `Chatter` app is **mostly** unchanged. We can copy the files `Chatt.kt` and `ChattStore.kt` over from lab0. With your lab3 open in Android Studio, open up your lab0. Copy (`ctl-c` on Windows or `cmd-c` on the Mac) `Chatt.kt` from your lab0's left pane and paste (`ctl-v` or `cmd-v`) to your lab3's left pane, **with `YOUR_PACKAGENAME` folder selected**. You should see a dialog box asking you to confirm the copy. Click `OK` (screenshot). Similarly copy `ChattStore` from your lab0 to your lab3. In both of these files, the first line containing the package name must be updated with `kotlinJpCChatter` :

```
package edu.umich.YOUR_UNIQNAME.kotlinJpCChatter
```

We only need to make one other change to `ChattStore` : find the line:

```
    val chatts = arrayListOf<Chatt?>()
```

and replace it with:

```
    val chatts = mutableStateListOf<Chatt>()
```

By declaring the `chatts` array to be of type `mutableStateListOf<Chatt>` . We make it a **published** `MutableState` version of `ArrayList<Chatt?>` . When a composable reads the value of a `MutableState` , its `RecomposeScope` is **automatically subcribed** to changes to the value. This means that whenever the value changes, a recomposition will be triggered for the composable. If the value is updated but the new value is the same as the old value, recomposition will not be triggered.

Performance consideration: when the value of a `MutableState` changes, all subscribing `RecomposeScope` s will recompose. Each composable is associated with a `RecomposeScope` . To prevent frequent recompositions of a View (i.e., composable) with many UI elements, try to isolate each UI element into a separate composable (function), thereby limiting changes to a `MutableState` to effect only Views that actually, truly, rely on the value of the `MutableState` .

ChattListRow

We want to display the `chatt` s retrieved from the backend in a timeline view. First we define what each row contains. Create another new Kotlin **File** (NOT "Class"), `ChattListRow` , and place the following Composable function in it:

```
@Composable
fun ChattListRow(index: Int, chatt: Chatt) {
    Column(modifier = Modifier.padding(8.dp, 0.dp, 8.dp, 0.dp)
        .background(color = Color(if (index % 2 == 0) 0xFFE0E0E0 else 0xFFEEEEEE))) {
        Row(horizontalArrangement = Arrangement.SpaceBetween, modifier=Modifier.fillMaxWidth(1f)) {
            chatt.username?.let { Text(it, fontSize = 17.sp, modifier = Modifier.padding(4.dp, 8.dp,

            chatt.timestamp?.let { Text(it, fontSize = 14.sp, textAlign = TextAlign.End, modifier =
        }

        chatt.message?.let { Text(it, fontSize = 17.sp, modifier = Modifier.padding(4.dp, 10.dp, 4.c
    }
}
```

> Text sizes are measured in **sp** (scale-independent-pixel) unit, which specifies font sizes relative to user's font-size preference—to assist visually-impaired users.

The content of a `ChattListRow` composable consists of a `Column` of two items: a `Row` on top and, below it, a text box containing the `chatt` message. The `chatt` message is padded so that the message is displayed inside the textbox away from the edges of the textbox.

The upper `Row` consists of two items: a text box containing the `username` and another text box containing the `timestamp` . The contents of these boxes are also padded except for their bottom edges. These two boxes are spaced out evenly with no extra spaces at the start and end of the row (which is what `Arrangement.SpaceBetween` mean), with the final result being that the two textboxes are flushed left and right respectively. By default, all textboxes take up as much space as needed by their contents.

▶ DSL

`MainView`

One of the main challenges in learning to develop apps declaratively is unfamiliarity with the platform's UI framework. What are all the available UI elements? How do we create them? What modifiers are there? What are their signatures? Everything is documented of course, but if you don't know what's available, if you don't know the name of the class to use, how do you search for it?

The Compose API Reference is the authoritative list of available APIs. Compose APIs are grouped into six groups, with the following four of immediate relevance to us:

- Compose Foundation
- Compose Runtime
- Compose UI, and

- Compose Material

Aside from the API Reference section listed at the top of each of the above pages, the change logs for the various versions are also useful to double the currency of APIs and their signatures when reading articles about Compose.

Create a new Kotlin **File** (NOT "Class"), `MainView`. The first thing we want to achieve in `MainView` is for the `chatt`s to be retrieved from the back-end server to populate the timeline when `MainView` is shown.

▶ File vs. Class

```
@Composable
fun MainView(context: Context, navController: NavHostController) {
    var isLaunching by rememberSaveable { mutableStateOf(true) }

    LaunchedEffect(Unit) {
        if (isLaunching) {
            isLaunching = false
            getChatts(context) {}
        }
    }
}
```

A composable is meant to be idempotent (can be called several times with the same outcome) and side-effect free. `LaunchedEffect()` allows us to call a function with side-effect when its `key` (first) argument changes. Setting the `key` to the unchanging `Unit` (or `true`) means `LaunchedEffect()` will be called only once in the composable lifecycle: when it's initially called, but not on re-compositions. Here "initial call" includes after a device configuration, such as orientation, change. To prevent the function being called again on orientation changes, we guard the call with an `isLaunching` variable whose value is saved across configuration changes (see the callout discussing `mutableStateOf()` and `rememberSaveable()` below).

Next we start declaring the `MainView` UI with the help of the `Scaffold()` composable. Add the following code **inside** your `MainView` composable right below the `LaunchedEffect()` block:

```
    Scaffold(
        topBar = {
            TopAppBar(title = {
                Text(
                    text = stringResource(R.string.chatter),
                    fontSize = 20.sp
                )
            })
        },
        floatingActionButton = {
            FloatingActionButton(
                backgroundColor = Color(0xFFFFC107),
                contentColor = Color(0xFF00FF00),
                modifier = Modifier.padding(0.dp, 0.dp, 8.dp, 8.dp),
                onClick = {
                    // navigate to PostView
                }
            ) {
                Icon(Icons.Default.Add, stringResource(R.string.post))
            }
        }
    ) {
        // content of Scaffold
            // describe the View
    }
```

Scaffold is a composable that implements the basic Material Design visual layout structure, providing slots for the most common top-level components such as topbar, floating action button, and others. By using Scaffold, we ensure the proper positioning of these components and that they interoperate smoothly. Scaffold and TopAppBar are examples of layout composables that follow the slot-based layout, a.k.a. Slot API pattern of Compose. In our case, we only want to add

1. a TopAppBar that consists of only a textbox containing the string Chatter and
2. a FloatingActionButton with a green '+' icon and a canary-colored background, offset 8dp from the trailing and bottom edges of the screen. We leave the onClick action of the floating action button empty for now.

Both the topBar and floatingActionButton parameters of Scaffold() take as argument a composable (function) with zero parameter and Unit return value. To use the TopAppBar() and FloatingActionButton() composables we thus need to wrap each in a parameterless lambda returning Unit value, as we have done above.

> 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").

Scaffold 's last parameter, content , also takes a composable as argument. Since it is the last argument, we have presented it as a trailing lambda in the above. Let's show the chatt timeline in this trailing lambda. Put the following code below the comment, //describe the View , in the trailing lambda:

```
        // content of Scaffold
            // describe the View
            LazyColumn(
                verticalArrangement = Arrangement.SpaceAround,
                modifier = Modifier.padding(0.dp, 10.dp, 0.dp, 0.dp)
            ) {
                items(count = chatts.size) { index ->
                    ChattListRow(index, chatts[index])
                }
            }
```

A LazyColumn is a Column that only allocates space and renders the view of currently visible data, ideal for large data set. The function items() apply the provided composable to a count of items, which in this case composes the view described by our composable ChattListRow() . We also specify that LazyColumn should spaced the items evenly with the spacing of the top and bottom edges being half the spacing between items (Arrangement.SpaceAround).

Swipe to refresh

Add this variable declaration to your `MainView` composable, **before** the call to `Scaffold()` :

```
    var isRefreshing by remember { mutableStateOf(false) }
```

then wrap the composeable `LazyRow()` inside the `SwipeRefresh()` composable such that the call to `SwipeRefresh()` becomes the full content of `Scaffold()` 's trailing lambda:

```
        // content of Scaffold
        SwipeRefresh(
            state = rememberSwipeRefreshState(isRefreshing),
            modifier = Modifier.background(color = Color(0xFFEFEFEF)),
            onRefresh = {
                getChatts(context){
                    isRefreshing = false
                }
            }
        ) {
            // describe the View
            LazyColumn(
                verticalArrangement = Arrangement.SpaceAround,
                modifier = Modifier.padding(0.dp, 10.dp, 0.dp, 0.dp)
                    .background(color = Color(0xFFEFEFEF))
            ) {
                items(count = chatts.size) { index ->
                    ChattListRow(index, chatts[index])
                }
            }
        }
```

Unlike the legacy `androidx.swiperefreshlayout` , `accompanist-swiperefresh` for compose only works if your swipe gesture starts from **inside** a list. If your list is empty, it is not shown on screen and you cannot initiate `SwipeRefresh` . You also cannot initiate `SwipeRefresh` from the empty space below a list. Otherwise, when the user swipes down on a list, `SwipeRefresh` shows a “loading” icon and run the `onRefresh` lambda expression. The variable `isRefreshing` is used to control when `SwipeRefresh` finishes running and stops showing the “loading” icon. `SwipeRefresh` requires `isRefreshing` to be a published state variable, which is why we declare it a `mutableStateOf()` here, with `false` as the initial value. Even though `isRefreshing` is a `MutableState` , it is declared inside a composable, which means that it is destroyed and recreated every time the composable is recomposed. To retain the value a `MutableState` **across recompositions**, we tag it with `remember {}` .

To publish a variable, we wrap it inside a `MutableState` type using `mutableStateOf()` or `mutableStateListOf()` . A composable that uses a published variable **automatically** subscribes to it. You don’t need to use `remember()` to subscribe to a published variable.

Use `remember()` only to retain states declared in composables **across recompositions**. This is normally used for view-logic states. States declared outside a composable are not destroyed by recomposition, though they are still destroyed by events impacting activity lifecycle, such as device-orientation change, for example. To save state across changes in orientation, use `rememberSaveable()` .

The lambda we provide to `onRefresh` in this case calls `getChatts()`. Upon completion, `getChatts()` sets `isRefreshing` to `false`, which will then be propagated to the `state` variable of `SwipeRefresh()` (saved across recompositions by `rememberSwipeRefreshState()`), causing the refresh to end. Note that `SwipeRefresh()` doesn't actually refresh the view, it calls `getChatts()` which refreshes the `chatts` array. Compose then recomposes the view given the updated `chatts` array.

> Cf. the call to `chattListAdapter.notifyDataSetChanged()` in conventional Android Views.

▶ by remember {}

`PostView`

We are not done with `MainView`, but let us put it aside for awhile and shift our focus to `PostView`, which we will use to compose and post a `chatt`. Create a new Kotlin file (not class), call it `PostView` and again create a composable with `Scaffold()`:

```kotlin
@Composable
fun PostView(context: Context, navController: NavHostController) {
    val username = stringResource(R.string.username)
    var message by rememberSaveable { mutableStateOf(context.getString(R.string.message)) }
    var enableSend by rememberSaveable { mutableStateOf(true) }

    Scaffold(
        // put the topBar here
    ) {
        Column(
            verticalArrangement = Arrangement.SpaceAround,
            modifier = Modifier.padding(8.dp, 0.dp, 8.dp, 0.dp).background(Color(0xffffffff))
        ) {
            Text(username, Modifier.padding(0.dp, 30.dp, 0.dp, 0.dp)
                .fillMaxWidth(1f), textAlign=TextAlign.Center, fontSize = 20.sp)
            TextField(
                value = message,
                onValueChange = { message = it },
                modifier = Modifier.padding(8.dp, 20.dp, 8.dp, 0.dp).fillMaxWidth(1f),
                textStyle = TextStyle(fontSize = 17.sp),
                colors = TextFieldDefaults.textFieldColors(backgroundColor=Color(0xffffffff))
            )
        }
    }
}
```

We describe `PostView` as a `Column` of two UI elements (as composables): a text box displaying the immutable variable `username`, an editable `TextField` whose:

1. `value` property is subscribed to the `MutableState` `message`,
2. `onValueChange` property will update `message` whenever the text inside the editable text box changes.

Because the `value` property is subscribed to `message`, whenever `onValueChange` updates `message`, `value` will be notified and will react, and recompose the view, thereby updating the message displayed (otherwise, text entered won't show up in the edit box).

The state `message` is declared `rememberSaveable { }` here to retain its value if the user navigates to `AudioView` and return prior to sending the `chatt`. As explained in a stackoverflow answer (second comment), "If you navigate to a different composable, the previous composable is destroyed - along with the state kept by `remember`. If you want to retain that state, `rememberSaveable` will preserve it when you navigate back. When you are navigating to and from screens, you are not recomposing but composing from a fresh start." Though `rememberSaveable { }` preserves state only if its composable is still on the navigation stack. Once its composable is popped off the stack, none of its states will be further retained.

Navigation between views

We use the Jetpack Navigation component to allow movements between composables. Recall that we have earlier added the dependency for `navigation-compose` to our Gradle build script.

To use Jetpack Navigation, first collect all the composables you want to put under navigation. Give each a name/identification in the form of a text string. This is known as the **path** to each composable (though the navigation "graph" is only one-evel deep) and will be used by Navigation to **route** to your composable.

You will need the use of a `NavController` to move from composable to composable. So that all composables under navigation have access to the `NavController`, you declare ("lift") it as high up in your composable hierarchy as necessary to encompass all composables under navigation as a subtree of the hierarchy at that point ("state hoisting"). For us, since we want to navigate between both of our composables (`MainView` and `PostView`), we declare the `NavController` at the root of the hierarchy, i.e., in `MainActivity`.

Associated with each `NavController` is a `NavHost`, which ties the `NavController` to the collection of composables you want to put under navigation.

`MainActivity`

Put the following code in the `MainActivity` class, in `MainActivity.kt` file created for you by Android Studio when you first created the `Empty Compose Project` earlier. Put the code after the call to `super.onCreate(savedInstanceState)`, replacing the template `setContent {}` provided:

```kotlin
        setContent {
            val navController = rememberNavController()
            NavHost(navController, startDestination = "MainView") {
                composable("MainView") {
                    MainView(this@MainActivity, navController)
                }
                composable("PostView") {
                    PostView(this@MainActivity, navController)
                }
            }
        }
```

We create a `NavController` using `rememberNavController()` and tie it to a `NavHost`. In creating the `NavHost`, we specify that navigation should start at the composable with name or path "MainView". Then we provide `NavHost` with all the composables under navigation. For each `composable()`, we first specify its path, as a `String`, and provide a lambda that calls the composable associated with the path. Note that we're providing each composable with the `Context` of `MainActivity`. This ensures that each View will have access to the app's theme.

We can pass an argument to destination composable when navigating to them, as we do with `AudioView` below. We can also nest one navigation graph inside another. Please see the Navigation documentation if interested.

As a general rule, a `@Composable` function can only be called by another `@Composable` function. The only exception is `setContent()`, which binds the given composable to the Activity as the root view of the Activity (`setContent()` replaces `setContentView()` of conventional Android Views). It is the only non-composable allowed to call a composable.

Once we have navigation setup, we can move between composables. Let's return to `MainView` and put in the code to navigate to `PostView` when the floating action button in clicked. Add the following line under the `// navigate to PostView` comment inside the lambda assigned to the `onClick` parameter of the call to `FloatingActionButton`:

```
                    // navigate to PostView
                    navController.navigate("PostView")
```

The `NavController` maintains a stack of visited composables. When you navigate from one composable to another, the `NavController` pushes the next composable on top of the previous ones on the stack.

With that, we're done with `MainView`! Let's finish up `PostView`.

When Android Studio created `MainActivity.kt` it also put in a `@Preview` block. As the name implies, the `@Preview` block allows you to preview your composables. The preview **only** renders your composables. It is not an emulator. It won't retrieve data from the back end to populate your composable. I found the preview to be of limited use and would just comment out the whole `@Preview` block, which automatically disables the preview and closes the Design (or Split) pane. The video, Compose Design Tools, shows you what is possible with `@Preview`.

Posting `chatts`

As we have done in `MainView`, we'd like to give the view a title, `Post`, in this case. Then we want a button at the upper right corner of `PostView` to post a `chatt`. Add the following code to `PostView` under the comment, `// put the topBar here` in the call to `Scaffold()`:

```
        // put the topBar here
        topBar = { TopAppBar(title = { Text(text = stringResource(R.string.post),
            fontSize=20.sp) },
            navigationIcon = {
                IconButton(onClick = { navController.popBackStack() } ) {
                    Icon(painter = painterResource(R.drawable.abc_vector_test),
                    stringResource(R.string.chatter))
                }
            },
        actions = { IconButton(onClick = {
            enableSend = false
            postChatt(context, Chatt(username, message))
            navController.popBackStack("MainView", inclusive = false)
        }, enabled = enableSend) {
            Icon(painter = painterResource(android.R.drawable.ic_menu_send), stringResource(R.st
        } }) }
```

In the call to create the `TopAppBar`, we give it a `title`, with font size 20 sp. The title is stored as a "string" resource named "post" in the `/app/res/strings.xml` file. Then we add a `navigationIcon` button, which is placed to the left of the title, following the default Material Design layout. We assign to the navigation button a "back arrow" icon. The actual asset name may be different depending on your theme, presently, the `abc_vector_test` vector asset (and no other vector asset) happens to have the "back arrow" icon we need.

Finally, we add an `action` button, which is placed flushed right on the top bar. We give the button a paper-plane icon (`ic_menu_send`) and name it, "Send". When the button is clicked, we call `postChatt()` with a `Chatt` object created from the values of `username` and `message` variables of the composable.

Once we've posted the `chatt`, we pop `PostView` off the navigation stack to get back to the `MainView` composable.

One last thing is to tell your new app to trust the self-signed certificate installed on your phone/emulator. Follow the instructions from lab0 on how to do this.

Congratulations! You've re-implemented lab0 from using Android Views to using Jetpack Compose. Try to build and run the code and confirm that it behaves the same as your Android Views version.

▶ Jetpack Compose version of Chatter

Preparing for lab4

The next lab will build on what you have done above (basically a Jetpack Compose version of lab0), so let's make a copy of it to prepare for the next lab:

- On your laptop, navigate to `YOUR_LABSFOLDER/`
- Make a copy of your `lab3` folder and name it **lab4** We'll keep `lab4` for the next lab, continue your work for the remainder of this lab in `lab3`.

Part II: Chatter with Audio

Setting up the back end

We first prepare the `Chatter` back end to support audio.

Install updates

Every time you ssh to your server, you will see something like:

```
N updates can be applied immediately.
```

if `N` is not 0, run the following:

```
server$ sudo apt update
server$ sudo apt upgrade
```

Failure to update your packages could lead to the lab back end not performing correctly and also make you vulnerable to security hacks.

If you see `*** System restart required ***` when you ssh to your server, please run:

```
server$ sync
server$ sudo reboot
```

Your ssh session will be ended at the server. Wait a few minutes for the system to reboot before you ssh to your server again.

Modified `Chatter` API data formats

In this lab, we allow user to post an audio clip with their `chatt` .

As in previous labs, the `chatt` s retrieval API will send back all accumulated chatts in the form of a JSON object consisting of a dictionary entry with key “chatts” and value being an array of string arrays. In addition to the three elements “username”, “message”, “timestamp”, each string array now carries one additional element which is a base64-encoded string of the audio clip:

```
{
    "chatts": [["username0", "message0", "timestamp0", "base64-encoded audio0"],
               ["username1", "message1", "timestamp1", "base64-encoded audio1"],
               ...
              ]
}
```

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

To post a `chatt` , the client correspondingly sends a JSON object consisting of “username”, “message”, and “audio”, where the value for key “audio” is a base64-encoded string. If no audio is posted, the value may be a JSON `null` or the empty string (`""`). For example:

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

Database table

As in previous labs, we first create a new table in our `chatterdb` database. Let's call this the `audio` table. It should have all the three columns of `username`, `message`, and `time` as in the `chatts` table of lab0. In addition, it should have an `audio` column of type `text`. Remember to give user `chatter` access to the new table.

> If you're not sure how to do any of the above, please review lab0 and lab1 back-end specs.

Editing `views.py`

Now, let's edit your `~/441/chatter/app/views.py` to handle audio uploads. Make a copy of your `postchatt()` function inside your `views.py` file and name the copy `postaudio()`. Add code to your `postaudio()` to extract the audio data from the JSON object and insert it into the `audio` table, along with the rest of its associated `chatt`. To the back-end database, the audio data is just a string. Note that your `INSERT` statement should target the `audio` table, not the `chatts` table.

Next, make a copy of your `getchatts()` function inside your `views.py` file and name the copy `getaudio()`. In `getaudio()`, replace the `chatts` table in the `SELECT` statement with the `audio` table. This statement will retrieve all data we need (including audio).

Save and exit `views.py`.

Routing for new urls

For the newly added `getaudio()` and `postaudio()` functions, add the following new routes to the `urlpatterns` array in `~/441/chatter/routing/urls.py`:

```
    path('getaudio/', views.getaudio, name='getaudio'),
    path('postaudio/', views.postaudio, name='postaudio'),
```

Remember to restart Gunicorn after you've updated `views.py` and `urls.py`.

Submitting your back end

- Commit new changes to the local repo with:

  ```
  server$ cd ~/441/chatter
  server$ git commit -m "lab3 back end"
  ```

  and push new changes to the remote GitHub repo with:

  ```
  server$ git push
  ```

- If `git push` fails due to new changes made to the remote repo, you will need to run `git pull` first. Then you may have to resolve any conflicts before you can `git push` again.

Once you are done with the back end, we'll move on to the front end.

Audio front end

In this section, we will go through the details about how to record, playback, pause, fast forward, rewind, stop, and upload audio. We will use the Android's built-in `MediaRecorder` to record from the device's microphone and save to local file, and `MediaPlayer` to play back audio on the device's speakers.

Requesting permission

To record audio, we need user's permission to access the device's microphone. Add this permission request to your `AndroidManifest.xml` file:

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

Now that we've added this permission tag, we'll be able to prompt user for permission later on.

Overview

To add support for audio in our app, we need to accomplish three things on the front end.

1. Work with Android's audio subsystem to record and playback audio.
2. Manage the transition of our audio frontend between recording, playback, and the various modes of the playback state: play, pause, stop, etc. We will use a state machine to track these transitions.
3. Update the UI elements corresponding to each state to control UX flow.

The first two tasks we will put in an `AudioPlayer` class, while the last task we will implement in `AudioView`, a composable.

Before we start implementing the `AudioPlayer`, let's add some resources we will be using in this app.

First, define some string constants we will be using. Add to `/app/res/values/strings.xml` the following resources:

```
<string name="back">Back</string>
<string name="audio">Audio</string>
<string name="doneButton">done</string>
<string name="rwndButton">rwnd</string>
<string name="ffwdButton">ffwd</string>
<string name="playButton">play</string>
<string name="stopButton">stop</string>
<string name="recButton">rec</string>
```

Next we add some icons to `/app/res/drawable`. On Android Studio's top menu bar choose `File > New > Vector Asset`. When the `Asset Studio` opens, click on the icon next to the `Clip Art:` field to bring up the

`Selection Icon` window. In the `Selection Icon` window, search for “pause”, click on the resulting `pause` icon, and click `OK` (screenshot). In the `Confirm Icon Path` screen, click `FINISH`. Repeat the process to add the following vector assets:

- exit to app
- forward 10
- mic
- mic none
- play arrow
- radio button checked
- replay 10
- share
- stop
- stop circle (outlined)

For `stop circle`, instead of “Filled”, choose “Outlined” in the drop-down menu (screenshot)

As in previous labs, we’ll collect all the extensions we’ll be using into one file. Create a new Kotlin file called `Extensions.kt` and put the same `toast()` extension to `Context` from the previous lab in it. Add the following `Color` property extensions to be used with open and filled mic:

```
val Color.Companion.OpenMic: Color
    get() = Color(0xFF8A9A5B)

val Color.Companion.FilledMic: Color
    get() = Color(0xFFB22222)
```

`AudioPlayer`

We will create an `AudioPlayer` class to interface with Android’s `MediaPlayer` and `MediaRecorder` subsystems. As the user uses the player to record or play audio, rewind and fast forward, pause and stop playback, we need to keep track of the state the player currently is at, to enable or disable some player control buttons. For example, we don’t want the user to be able to press the play button in the middle of recording. The state machine capturing all the states of the player and the events triggering transition between states is shown in Figure 1:

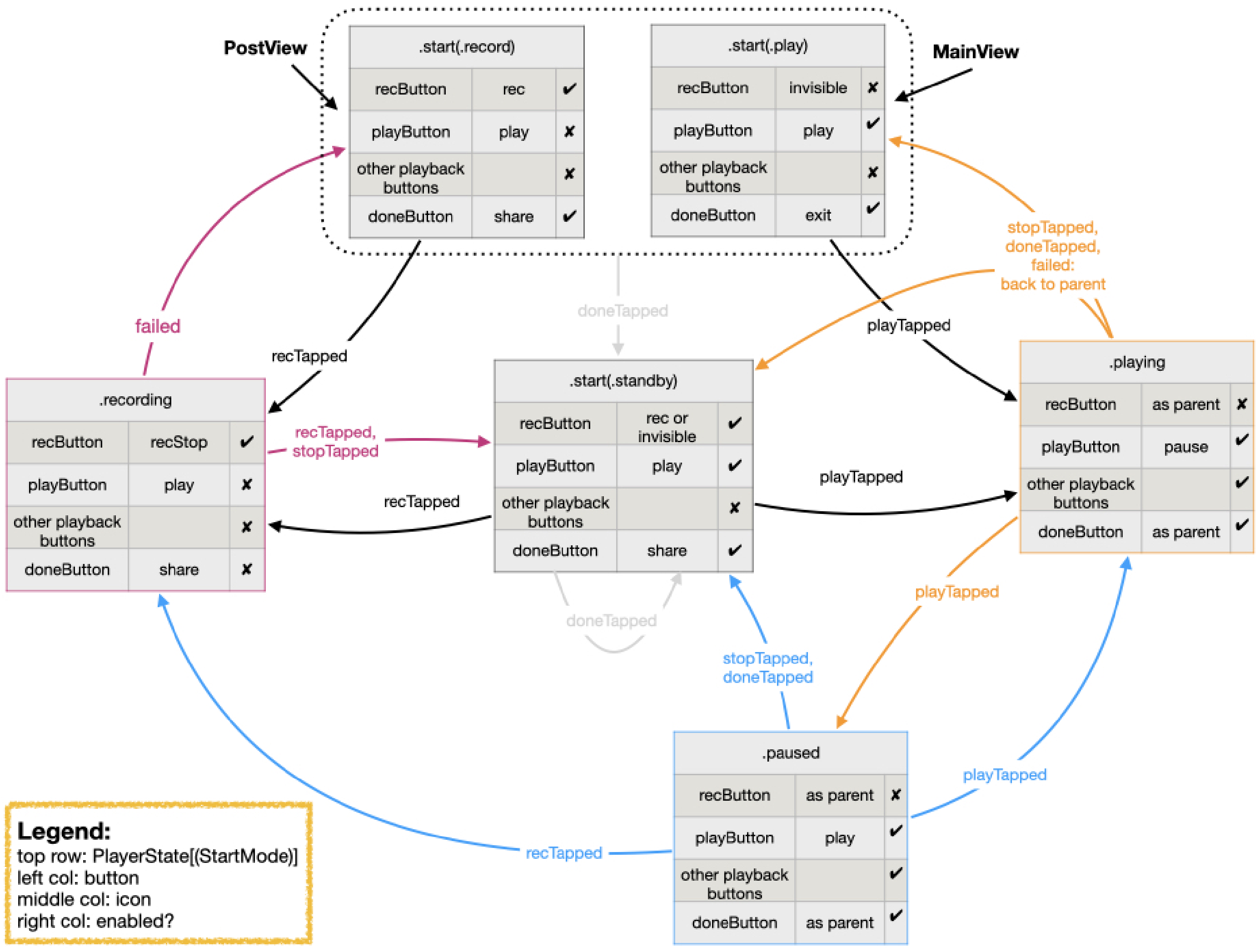


Figure 1: AudioPlayer State Machine

Create a new Kotlin file (not class) and call it `AudioPlayer`. We first define the `PlayerState` s a player can be in:

```
enum class StartMode {
    standby, record, play
}
sealed class PlayerState {
    class start(val mode: StartMode): PlayerState()
    object recording: PlayerState()
    class playing(val parent: StartMode): PlayerState()
    class paused(val grand: StartMode): PlayerState()
}
```

As shown in Figure 1, the `start` state consists of three modes, which we have implemented using an `enum`. The `PlayerState` itself is implemented as a `sealed class` to give us more flexibility in defining the possible cases. (Please see the lecture on `enum` and `sealed class` for further explanation of these.)

To implement the transition between states as shown in Figure 1, we enumerate the possible transition events:

```
enum class TransEvent {
    recTapped, playTapped, stopTapped, failed
}
```

and encode the state transitions in a `transition()` method of the `PlayerState` sealed class:

```
    fun transition(event: TransEvent): PlayerState {
        return when (this) {
            is start -> when (mode) {
                StartMode.record -> if (event == TransEvent.recTapped) recording else this
                StartMode.play -> if (event == TransEvent.playTapped) playing(StartMode.play) else t
                StartMode.standby -> when (event) {
                    TransEvent.recTapped -> recording
                    TransEvent.playTapped -> playing(StartMode.standby)
                    else -> this
                }
            }
            recording -> when (event) {
                TransEvent.recTapped, TransEvent.stopTapped -> start(StartMode.standby)
                TransEvent.failed -> start(StartMode.record)
                else -> this
            }
            is playing -> when (event) {
                TransEvent.playTapped -> paused(this.parent)
                TransEvent.stopTapped, TransEvent.failed -> start(this.parent)
                else -> this
            }
            is paused -> when (event) {
                TransEvent.recTapped -> recording
                TransEvent.playTapped -> playing(this.grand)
                TransEvent.stopTapped -> start(StartMode.standby)
                else -> this
            }
        }
    }
```

You can trace the transitions encoded in this method by following along the edges of the state machine diagram in Figure 1.

As for the `AudioPlayer` class, we start with the following declaration:

```
class AudioPlayer() {
    var playerState by mutableStateOf<PlayerState>(PlayerState.start(StartMode.standby))
    var audio = ByteArray(0)

    private lateinit var audioFilePath: String
    private lateinit var mediaPlayer: MediaPlayer
    private lateinit var mediaRecorder: MediaRecorder

    constructor(context: Context, extFilePath: String): this() {
        audioFilePath = extFilePath
        mediaPlayer = MediaPlayer()
        // API Level 31: MediaRecorder needs context
        mediaRecorder = MediaRecorder(/*context*/)
        // Otherwise ignore Android Studio's warning that context is never used
    }
}
```

The class property `audio` will hold the audio data recorded or to be played. Both the `MediaRecorder` and `MediaPlayer` only work with stored audio. We create an instance of `MediaPlayer` and `MediaRecorder` and `audioFilePath` in a secondary constructor. We call this secondary constructor in `MainActivity` to create the `AudioPlayer` initially. These properties only need to be initialized on first construction, not on subsequent restoration on device orientation change. The property `audioFilePath` **must be** initialized to point to a temporary audio file in Android's external cache directory.

The property `playerState` that holds the player's current state is declared as a `mutableStateOf()`, which means that it is a published state that can be subscribed to by composables. (Please see notes on observer pattern for explanation of these concepts.) Upon instantiation, `AudioPlayer` starts in the `standby` mode of its `start` state, as is reflected in the initial value of `playerState`.

From the `standby` mode, the player can be put in `record` or `play` mode. Add the following `AudioPlayer` methods:

```
    fun setupRecorder() {
        playerState = PlayerState.start(StartMode.record)
        audio = ByteArray(0)

        mediaPlayer.reset()
    }

    fun setupPlayer(audioStr: String) {
        playerState = PlayerState.start(StartMode.play)
        audio = Base64.decode(audioStr, Base64.DEFAULT)

        mediaPlayer.reset()
        preparePlayer()
    }
```

When the player is set to `record` mode, we always re-initialize `audio` to an empty byte array so as not to accidentally carry over previously recorded or played audio. When the player is set to `play` mode, we expect to be passed along a Base64-encoded string to be played back. This would normally be an audio clip associated with a posted `chatt`. We store the decoded string in the `audio` property and prepare the `MediaPlayer` for playback. In both cases, we reset the `MediaPlayer` to clear it of any audio clip from a previous use of `AudioView`.

`MediaPlayer`

It may be useful to consult the MediaPlayer State Diagram found in Android's MediaPlayer documentation as you implement the playback function.

Add the following `preparePlayer()` method to the `AudioPlayer` class:

```
    private fun preparePlayer() {
        mediaPlayer.setOnCompletionListener {
            playerState = playerState.transition(TransEvent.stopTapped)
        }

        val os: OutputStream = try { FileOutputStream(audioFilePath) } catch (e: IOException) {
            Log.e("preparePlayer: ", e.localizedMessage ?: "IOException")
            return
        }
        os.write(audio)
        os.close()

        with (mediaPlayer) {
            setDataSource(audioFilePath)
            setVolume(1.0f, 1.0f) // 0.0 to 1.0 raw scalar
            prepare()
        }
    }
```

To prepare the player, we first specify a listener for when the player finishes playing. In this case, we call the `transition()` method of `PlayerState` to transition `playerState` to the appropriate state (as shown in Figure 1) as if the `stop` button has been pressed.

We then write the `audio` clip to a temporary file stored in `audioFilePath`. We pass this file path to `MediaPlayer` to play back. We set the play back volume and call the `prepare()` method of `MediaPlayer`.

With the `MediaPlayer` setup done, we now fill in the code to perform the action associated with playback buttons. Create a `playTapped()` method in `AudioPlayer` class as follows:

```
    fun playTapped() {
        with (mediaPlayer) {
            if (playerState is PlayerState.playing) {
                pause()
            } else {
                this.start()
            }
        }
        playerState = playerState.transition(TransEvent.playTapped)
    }
```

If the play button is tapped when the player is playing, the user wants to pause play, call `MediaPlayer`'s `pause()` method. Otherwise, if the player is not playing when the play button is tapped, call `MediaPlayer`'s `start()` method which will either start or resume play. According to Figure 1, if playback is paused by tapping the play button a second time, the state is changed to "paused", tapping the play button again will then resume play. In all cases, we transition the `playerState` to the appropriate next state by calling the `transition()` method of `PlayerState`.

▶ MediaPlayer error messages

If the stop button is tapped, we also only pause playback instead of stopping it. According to the MediaPlayer state diagram, once the `MediaPlayer` is stopped, we can't restart play back without preparing the player again (which could throw an IO exception that needs to be caught). So instead, we simply reset the play head to the beginning of the audio clip and call `transition()` to transition `playerState` appropriately:

```
    fun stopTapped() {
        mediaPlayer.pause()
        mediaPlayer.seekTo(0)
        playerState = playerState.transition(TransEvent.stopTapped)
    }
```

The final two playback controls simply move the play head forward or backward by 10 seconds, without changing state in the `StateMachine`. Add these to your `AudioPlayer` class:

```
    fun ffwdTapped() {
        mediaPlayer.seekTo(mediaPlayer.currentPosition+10000)
    }

    fun rwndTapped() {
        mediaPlayer.seekTo(mediaPlayer.currentPosition-15000)
    }
```

MediaRecorder

It may be useful to consult the MediaRecorder State Diagram found in Android's MediaRecorder documentation as you implement the recording function.

Earlier we have defined the method `setupRecorder()`. Once the recorder is set up, recording is initiated by the user tapping on a record button, which calls the `recTapped()` method of `AudioPlayer`:

```
    fun recTapped() {
        if (playerState == PlayerState.recording) {
            finishRecording()
        } else {
            startRecording()
        }
    }
```

When the recording button is tapped, if we are currently recording, tapping the record button again stops the recording by calling the `finishRecording()` method of `AudioPlayer`. If we are, however, not currently recording, we start recording by calling the `startRecording()` method of the `AudioPlayer` class:

```
    private fun startRecording() {
        // reset player because we'll be re-using the output file that may have been primed at the p
        mediaPlayer.reset()

        playerState = playerState.transition(TransEvent.recTapped)

        with (mediaRecorder) {
            setAudioSource(MediaRecorder.AudioSource.MIC)
            setOutputFormat(MediaRecorder.OutputFormat.MPEG_4)
            setAudioEncoder(MediaRecorder.AudioEncoder.AAC)
            setOutputFile(audioFilePath)
            try {
                prepare()
            } catch (e: IOException) {
                Log.e("startRecording: ", e.localizedMessage ?: "IOException")
                return
            }
            this.start()
        }
    }
```

To start recording, we first ensure that the `MediaPlayer` is not otherwise using the temporary audio file we will be using. We do this by calling the `reset()` method of the `MediaPlayer`. Next we call the `transition()` method of `PlayerState` to put the player in the correct state. Next we prepare the `MediaRecorder`: specifying the audio source (mic), output format (mpeg4), audio encoder (AAC), and output file (`audioFilePath`). Then we call the `prepare()` method of `MediaRecorder` and start recording by calling the `start()` method of `MediaRecorder`.

If `recTapped()` is called when recording is ongoing, we end recording by calling the `finishRecording()` method of `AudioPlayer`:

```
    private fun finishRecording() {
        mediaRecorder.stop()
        mediaRecorder.reset()
        try {
            var read: Int
            audio = ByteArray(65536)
            val fis = FileInputStream(audioFilePath)
            val bos = ByteArrayOutputStream()
            while (fis.read(audio, 0, audio.size).also { read = it } != -1) {
                bos.write(audio, 0, read)
            }
            audio = bos.toByteArray()
            bos.close()
            fis.close()
        } catch (e: IOException) {
            Log.e("finishRecording: ", e.localizedMessage ?: "IOException")
            playerState = playerState.transition(TransEvent.failed)
            return
        }
        playerState = playerState.transition(TransEvent.recTapped)
        preparePlayer()
    }
```

The function stops the recording, load the recorded clip into the `audio` property (to be uploaded to the `Chatter` back end along with the posted `chatt`), calls `transition()` to perform the appropriate state transition, and prepare the `MediaPlayer` in case the user wants to play back the recorded audio before posting it.

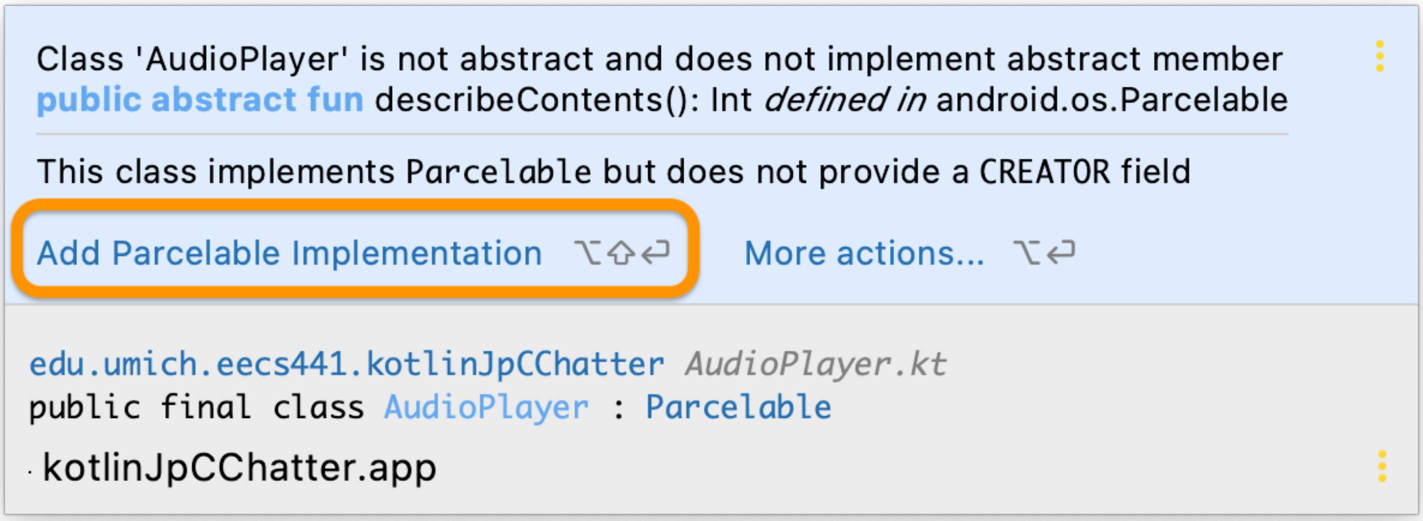
Once the user is satisfied with the recording, they tap the done button, which calls the `doneTapped()` method to reset both the `MediaPlayer` and `MediaRecorder`:

```
    fun doneTapped() {
        if (playerState == PlayerState.recording) {
            finishRecording()
        } else {
            mediaRecorder.reset()
        }
        mediaPlayer.start() // so that playback works on revisit
        stopTapped()
    }
```

With that, we are done with the `AudioPlayer`! We have one more thing to do though. So that the player can be saved across screen orientations changes, we need to make it `Parcelable`. Change the declaration of `AudioPlayer` to implement the `Parcelable` interface:

```
class AudioPlayer(): Parcelable {
```

Android Studio will underline `class AudioPlayer` as erroneous because it's missing the implementation of the protocol. Hover over the class name until the error dialog box pops up and then select `Add Parcelable Implementation`:


Class 'AudioPlayer' is not abstract and does not implement abstract member public abstract fun describeContents(): Int *defined in* android.os.Parcelable

This class implements Parcelable but does not provide a CREATOR field

edu.umich.eecs441.kotlinJpCChatter *AudioPlayer.kt*
public final class AudioPlayer : Parcelable
kotlinJpCChatter.app

Replace the generated `constructor(parcel: Parcel) : this() { ... }` block with:

```
    constructor(parcel: Parcel) : this() {
        audio = parcel.createByteArray() ?: ByteArray(0)
        audioFilePath = parcel.readString()!!
    }
```

> See Implementing the Parcelable Interface in Android if you're interested in creating `Parcelable` manually.

Now we declare the UI to go along with the audio player.

`AudioView`

The UI for our audio player consists of buttons one would expect to find in an audio player: record, play, stop, rewind, and fast forward. In addition, we also have a "done" button for when the user is done and wish to exit the audio player. As shown in Figure 1, the state the player is in determines whether the record button is shown, whether it is shown in red or in green, whether the play button shows the "play" or "pause" icon, etc. In "playing" and "paused" states, whether the record button is shown further depends on the state the player was at prior to arriving in these states.

We will use a `PlayerUIState` object to hold the modifiers affecting the appearance of every UI element in `AudioView`. Whenever the audio player state changes, `AudioView` will propagate the change to all UI element modifiers in `PlayerUIState` by invoking its `propagate()` method. The UI elements all subscribe to `PlayerUIState`, so whenever `propagate()` updates a property of `PlayerUIState`, the affected UI element will be recomposed. No child-composable of `AudioView` subscribes to `AudioPlayer.playerState` directly.

▶ Design rationale and alternatives

Let's start by creating a new Kotlin file, `AudioView` and put the following `PlayerUIState` in this file as a helper type of `AudioView`:

```
class PlayerUIState(): Parcelable {
    var recVisible = true
    var recEnabled = true
    var recColor = Color.Black
    var recIcon = R.drawable.ic_baseline_radio_button_checked_24 // initial value

    var playCtlEnabled = false
    var playCtlColor = Color.LightGray

    var playEnabled = false
    var playColor = Color.LightGray
    var playIcon = R.drawable.ic_baseline_play_arrow_24 // initial value

    var doneEnabled = true
    var doneColor = Color.DarkGray
    var doneIcon = R.drawable.ic_baseline_share_24 // initial value

    private fun playCtlEnabled(enabled: Boolean) {
        playCtlEnabled = enabled
        playCtlColor = if (enabled) Color.DarkGray else Color.LightGray
    }

    private fun playEnabled(enabled: Boolean) {
        playIcon = R.drawable.ic_baseline_play_arrow_24
        playEnabled = enabled
        playColor = if (enabled) Color.DarkGray else Color.LightGray
    }

    private fun pauseEnabled(enabled: Boolean) {
        playIcon = R.drawable.ic_baseline_pause_24
        playEnabled = enabled
        playColor = if (enabled) Color.DarkGray else Color.LightGray
    }

    private fun recEnabled() {
        recIcon = R.drawable.ic_baseline_radio_button_checked_24
        recEnabled = true
        recColor = Color.Black
    }

    private fun doneEnabled(enabled: Boolean) {
        doneEnabled = enabled
        doneColor = if (enabled) Color.DarkGray else Color.LightGray
    }

    fun propagate(playerState: PlayerState) = when (playerState) {
        is PlayerState.start -> {
            when (playerState.mode) {
                StartMode.play -> {
                    recVisible = false
                    recEnabled = false
                    recColor = Color.Transparent
                    playEnabled(true)
                    playCtlEnabled(false)
                    doneIcon = R.drawable.ic_baseline_exit_to_app_24
                    doneColor = Color.DarkGray
                }
                StartMode.standby -> {
                    if (recVisible) recEnabled()
                    playEnabled(true)
                    playCtlEnabled(false)
                    doneEnabled(true)
                }
                StartMode.record -> {
                    // initial values already set up for record start mode.
                }
            }
        }
        PlayerState.recording -> {
            recIcon = R.drawable.ic_outline_stop_circle_24
            recColor = Color.FilledMic
            playEnabled(false)
            playCtlEnabled(false)
            doneEnabled(false)
        }
        is PlayerState.paused -> {
            if (recVisible) recEnabled()
            playIcon = R.drawable.ic_baseline_play_arrow_24
        }
        is PlayerState.playing -> {
            if (recVisible) {
                recEnabled = false
                recColor = Color.LightGray
            }
            pauseEnabled(true)
            playCtlEnabled(true)
        }
    }
}
```

As when we turned `AudioPlayer` parcelable, Android Studio will underline `class PlayerUIState` as erroneous due to lack of `Parcelable` implementation. Hover over the `PlayerUIState` class name and click `Add Parcelable Implementation` when the error dialog box pops up. (We actually only need to save the `recVisible` property across orientation changes because the others are populated by `propagate()` in an idempotent manner.)

Now that we have primed all the states affecting the UI for the reactive framework, we can declare the UI (composable) for the audio player, `AudioView`. In `AudioView` we first construct an instance of `PlayerUIState` to hold the modifiers of its UI elements. Because we want the children composables of `AudioView` to subscribe to this instance of `PlayerUIState`, we declare it as `mutableStateOf()`. Further, so that we maintain the same `PlayerUIState` across recompositions of `AudioView` and across screen orientation changes, we use `rememberSaveable { }` with this instantiation of `PlayerUIState`.

Since `AudioView` uses `AudioPlayer.playerState`, it is automatically subscribed to it. Everytime `playerState` is updated, `AudioView` will be recomposed. Upon every (re)composition of `AudioView`, we call the `propagate()` method of `PlayerUIState`, to update the modifiers of the UI elements according to the current value of `playerState`. Add the following code to your `AudioView.kt`:

```
@Composable
fun AudioView(navController: NavHostController, audioPlayer: AudioPlayer, autoPlay: Boolean?) {
    val playerUIState by rememberSaveable { mutableStateOf(PlayerUIState()) }
    var isLaunching by rememberSaveable { mutableStateOf(true) }

    playerUIState.propagate(audioPlayer.playerState)
    LaunchedEffect(Unit) {
        if (isLaunching) {
            isLaunching = false
            if (autoPlay ?: false) {
                audioPlayer.playTapped()
            }
        }
    }
}
```

`AudioView()` also takes a third argument, `autoPlay`. When `autoPlay` is `true`, `AudioView` automatically invokes `AudioPlayer.playTapped()`. Recall that a composable is meant to be idempotent (can be called several times with the same outcome) and side-effect free. Similar to how we have done in `MainView` earlier, we use `LaunchedEffect()` to call `playTapped()` upon the initial run of `AudioView()`. We similarly guard the call with an `isLaunching` variable whose value is saved across configuration changes so that `playTapped()` is not called on orientation changes.

With `playerUIState` and `LaunchedEffect()` set up, we can now lay out the audio player screen. Add the following to your `AudioView()` composable right below the `LaunchedEffect()` block:

```
    Column(verticalArrangement = Arrangement.SpaceBetween,
        horizontalAlignment = Alignment.CenterHorizontally,
        modifier=Modifier.fillMaxHeight(1f)) {
        Spacer(modifier=Modifier.fillMaxHeight(.1f))
        Row(horizontalArrangement = Arrangement.SpaceEvenly, modifier=Modifier.fillMaxWidth(1f)) {
            StopButton(audioPlayer, playerUIState)
            RwndButton(audioPlayer, playerUIState)
            PlayButton(audioPlayer, playerUIState)
            FfwdButton(audioPlayer, playerUIState)
            DoneButton(navController, audioPlayer, playerUIState)
        }
        RecButton(audioPlayer, playerUIState)
    }
```

We now add the definition of the `RecButton()` composable, outside the `AudioView` composable but in the same file:

```
@Composable
fun RecButton(audioPlayer: AudioPlayer, playerUIState: PlayerUIState) {
    Button(onClick = { audioPlayer.recTapped() },
        enabled = playerUIState.recEnabled,
        modifier = Modifier.fillMaxSize(.5f),
        colors = ButtonDefaults.buttonColors(backgroundColor = Color.White,
            disabledBackgroundColor = Color.White),
        elevation = ButtonDefaults.elevation(0.dp)
    ) {
        Icon(painter = painterResource(playerUIState.recIcon),
            modifier=Modifier.scale(4f).weight(weight = .8f),
            contentDescription = stringResource(R.string.recButton),
            tint = playerUIState.recColor
        )
    }
}
```

When the record button is tapped, it calls `audioPlayer.recTapped()`. Whether the button is enabled depends on the value in `playerUIState.recEnabled`. The color of the button depends on the value in `playerUIState.recColor`.

Here's the definition of the `DoneButton()` composable:

```
@Composable
fun DoneButton(navController: NavHostController, audioPlayer: AudioPlayer, playerUIState: PlayerUISt
    Button(onClick = {
        audioPlayer.doneTapped()
        navController.popBackStack()
    },
        enabled = playerUIState.doneEnabled,
        colors = ButtonDefaults.buttonColors(backgroundColor = Color.White,
            disabledBackgroundColor = Color.White),
        elevation = ButtonDefaults.elevation(0.dp)
    ) {
        Icon(painter = painterResource(playerUIState.doneIcon),
            modifier=Modifier.scale(1.7f).padding(end=8.dp),
            contentDescription = stringResource(R.string.doneButton),
            tint = playerUIState.doneColor
        )
    }
}
```

When the done button is tapped, in addition to calling `audioPlayer.doneTapped()`, we also call `navController.popBackStack()` to pop the navigation stack back to the composable that launched `AudioView`.

The other buttons are defined similarly to the `doneButton` except they don't pop the navigation stack. Go ahead and implement the other buttons.

There is one more thing we need to take care of. User could leave the `AudioView()` composable by tapping the `Done` button or by tapping the device's `Back Button` (◀). In the latter case, we also want to stop play back or recording before leaving the view. Go back to your `AudioView()` composable and add the following call to the `BackHandler()` composable **inside** `AudioView()`, below the `Column()` block:

```
    BackHandler {
        audioPlayer.doneTapped()
        navController.popBackStack()
    }
```

With that, we're done with the audio player and its view! We now need to integrate it with the rest of `Chatter`.

Chatt

Let's start with our Model. In `Chatt.kt`, the `Chatt` class would need to hold an extra audio `String`:

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

The audio property uses the same `ChattPropDelegate` from lab1. Copy the property delegate from lab1 to `Chatt.kt`.

ChattStore

To retrieve the audio `String`, replace `getchatts` with `getaudio` in the url construction for `getRequest`. Then modify the `if (chattEntry.length() == nFields) {` block of the `getChatts()` method in `ChattStore` to read:

```
                    chatts.add(Chatt(username = chattEntry[0].toString(),
                        message = chattEntry[1].toString(),
                        timestamp = chattEntry[2].toString(),
                        audio = chattEntry[3].toString()
                    ))
```

And modify the declaration of `jsonObj` in the `postChatt` method to read:

```
        val jsonObj = mapOf(
            "username" to chatt.username,
            "message" to chatt.message,
            "audio" to chatt.audio
        )
```

Also change the declaration of `postRequest` to read:

```
        val postRequest = JsonObjectRequest(Request.Method.POST,
            serverUrl+"postaudio/", JSONObject(jsonObj),
            {
                Log.d("postChatt", "chatt posted!")
                getChatts(context){}
            },
            { error -> Log.e("postChatt", error.localizedMessage ?: "JsonObjectRequest error") }
        )
```

The changes here are:

1. to use `postaudio` instead of `postchatt` in the POST url, and
2. call `getChatts()` here to retrieve an updated list of `chatt` s from the back end. We take advantage of reactive update of `MainView` to have it display an updated timeline automatically without the user having to swipe to refresh. Recall that thanks to reactive UI, `MainView` will update automatically when the `chatt` array in `ChattStore` is updated.

That's all the modifications we need to make to the Model. We now turn to integrating audio to our View:

`MainActivity`

Add the following property to your `MainActivity` class:

```
    private lateinit var audioPlayer: AudioPlayer
```

and add the following method to the class to save `audioPlayer` across screen orientation changes:

```
    override fun onSaveInstanceState(savedInstanceState: Bundle) {
        super.onSaveInstanceState(savedInstanceState)
        savedInstanceState.putParcelable("AUDIOPLAYER", audioPlayer)
    }
```

In the `onCreate()` method of `MainActivity` , set up an Android's `ActivityResultContracts` to prompt user for permission to access the mic. The name of the contract is `RequestPermission` (singular). Once the contract is created, register it with the following callback handler, in the form of a lambda expression:

```
 { granted ->
     if (!granted) {
         toast("Audio access denied")
         finish()
     }
 }
```

Since we have no further use for the contract and registered launcher beyond `onCreate()` , you can launch it immediately with `Manifest.permission.RECORD_AUDIO` as the launch argument.

Next, right after the permission request, if we've previously saved `audioPlayer` as an instance state prior to orientation change, retrieve and restore it. Otherwise, create a temporary file to hold recorded audio, or audio to be played back, and create a new instance of `AudioPlayer`:

```
savedInstanceState?.run {
    audioPlayer = getParcelable("AUDIOPLAYER")!!
} ?: externalCacheDir?.let {
    audioPlayer = AudioPlayer(this, "${it.absolutePath}/chatteraudio.m4a")
} ?: run {
    Log.e("AudioActivity", "external cache dir null")
    toast("Cannot create temporary file!")
    finish()
}
```

▶ onRestoreInstanceState()

To the existing `NavHost` composable calls to `MainView` and `PostView`, add a third parameter, `audioPlayer`, and add a new `composable()` entry for `AudioView`. Your `NavHost(){ }` block should then look like this:

```
NavHost(navController, startDestination = "MainView") {
    composable("MainView") {
        MainView(this@MainActivity, navController, audioPlayer)
    }
    composable("PostView") {
        PostView(this@MainActivity, navController, audioPlayer)
    }
    // passing an optional, nullable argument
    composable("AudioView?autoPlay={autoPlay}",
        arguments = listOf(navArgument("autoPlay") {
            type = NavType.BoolType
            defaultValue = false
        })) {
        AudioView(navController, audioPlayer, it.arguments?.getBoolean("autoPlay"))
    }
}
```

The "path" for `AudioView` says that it takes an optional (`?`) argument of type `NavType.BoolType`, with default value set to `false`.

`MainView`

The are three changes we need to make to `MainView`:

1. add a third parameter to the composable definition:

   ```
   @Composable
   fun MainView(context: Context, navController: NavHostController, audioPlayer: AudioPlayer) {
   ```

2. pass the `audioPlayer` on to `ChattListRow`:

   ```
                   ChattListRow(index, chatts[index], navController, audioPlayer)
   ```

3. and set up `audioPlayer` for recording when the button that launches `PostView` is clicked. Find:

```
                onClick = {
                    navController.navigate("PostView")
                }
```

and replace it with:

```
                onClick = {
                    audioPlayer.setupRecorder()
                    navController.navigate("PostView")
                }
```

ChattListRow

Modify `ChattListRow`'s signature to receive a `navController` and `audioPlayer` as third and fourth arguments:

```
@Composable
fun ChattListRow(index: Int, chatt: Chatt, navController: NavHostController, audioPlayer: AudioPlaye
```

If there's an audio clip associated with the `chatt`, wrap the text box displaying the `chatt` message:

```
        chatt.message?.let { Text(it, fontSize = 17.sp, modifier = Modifier.padding(4.dp, 10.dp, 4.c
```

in a `Row()` composable along with a button to launch the `AudioView`:

```
        Row(horizontalArrangement = Arrangement.SpaceBetween,
            modifier=Modifier.fillMaxWidth(1f)) {
            chatt.message?.let { Text(it, fontSize = 17.sp, modifier = Modifier.padding(4.dp, 10.dp,
            chatt.audio?.let {
                IconButton(onClick = {
                    audioPlayer.setupPlayer(it)
                    navController.navigate("AudioView?autoPlay=true")
                },
                    modifier = Modifier.padding(0.dp, 0.dp, 8.dp, 0.dp).align(Alignment.Bottom)) {
                    Icon(painter = painterResource(android.R.drawable.stat_notify_voicemail),
                        contentDescription = stringResource(R.string.audio),
                        modifier = Modifier.scale(1.4f),
                        tint = Color.DarkGray
                    )
                }
            }
        }
```

Incidentally, we also set up `audioPlayer` to play back the audio clip when the button that launches `AudioView` is clicked.

PostView

As with `MainView`, the first thing we need to do is to add a third argument to `PostView`:

```
@Composable
fun PostView(context: Context, navController: NavHostController, audioPlayer: AudioPlayer) {
```

In the call to `postChatt()`, we now add the audio element to the `chatt` to be posted, after encoding it into a base64 string:

```
                postChatt(context, Chatt(username, message,
                    audio = Base64.encodeToString(audioPlayer.audio, Base64.DEFAULT)))
```

Finally, wrap the existing `TextField()` holding the `chatt` message in a `Row()` composable along with a button to launch `AudioView`:

```
            Row(horizontalArrangement = Arrangement.SpaceBetween, modifier=Modifier.fillMaxWidth(1f)
                TextField(
                    value = message,
                    onValueChange = { message = it },
                    modifier = Modifier.padding(8.dp, 20.dp, 8.dp, 0.dp).fillMaxWidth(.8f),
                    textStyle = TextStyle(fontSize = 17.sp),
                    colors = TextFieldDefaults.textFieldColors(backgroundColor = Color.Transparent)
                )
                IconButton(
                    onClick = {
                        navController.navigate("AudioView")
                    },
                    modifier = Modifier.padding(end = 4.dp).align(Alignment.Bottom)
                ) {
                    if (audioPlayer.audio.size == 0) {
                        Icon(painter = painterResource(R.drawable.ic_baseline_mic_none_24),
                            contentDescription = stringResource(R.string.audio),
                            modifier = Modifier.scale(1.8f),
                            tint = Color.OpenMic
                        )
                    } else {
                        Icon(painter = painterResource(R.drawable.ic_baseline_mic_24),
                            contentDescription = stringResource(R.string.audio),
                            modifier = Modifier.scale(1.8f),
                            tint = Color.FilledMic
                        )
                    }
                }
            }
```

Congratulations, you've successfully integrated audio into your `Chatter` app written in Jetpack Compose!

To use the Android emulator to work with audio, follow the instructions in our Getting Started with Android Development.

Submission guidelines

> **IMPORTANT:** If you work in team, remember to put your team mate's uniqnames in lab3 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 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 lab3 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 the GitHub website to confirm that your project files for lab3 have been uploaded to your GitHub repo under folder `lab3`.

References

Jetpack Compose

Concepts

- Understanding Jetpack Compose
- Thinking in Compose
- Architecting your Compose UI
- Observer vs Pub-Sub pattern
- Layouts in Compose
- Slot API
- Lists
- State and Jetpack Compose
- Navigating with Compose
- Get started with Jetpack Compose

Documentations

- Use Android Studio with Jetpack Compose
- androidx.compose
- Scaffold
- Column
- LazyColumn
- Button
- MutableState
- mutableStateOf()
- mutableStateListOf()
- remember
- rememberSaveable
- Navigation
    - Navigate with arguments **Errata**: `nullability = true` should be `nullable = true`.
- NavController
- NavHost

3rd-party articles on Jetpack Compose

⚠ Depending on date of publication, Compose APIs used in 3rd-party articles may have been deprecated or their signatures may have changed. Always consult the authoritative official documentation and change logs for the most up to date version.

Layout and Components

- Using Jetpack Compose Scaffold as Template
- Using the Scaffold and Slot APIs in Jetpack Compose
- Scaffold Widgets Template
- TopAppBar
- RecyclerView and LazyColumnFor in Jetpack Compose
- Swipe Refresh layout for Jetpack Compose
- Jetpack Compose Components
- Row
- Column
- Navigating in Jetpack Compose
- Passing multi typed data between screens in Jetpack Compose navigation component
- ConstraintLayout loves Jetpack Compose
- Jetpack Compose’s Rows and Colums or ConstraintLayout?
- The power of @Preview

State

- Recomposition Made Easy
- Remember Made Easy
- Finite State Machine as a ViewModel for Jetpack Compose Screens

Side-effects

- Jetpack Compose Effect Handlers
- Side-effects in Compose
- Handling the system back button

Themes and Styles

- Theme Made Easy
- Setting up Themes
- Theming in Jetpack Compose
- How to create a truly custom theme in Jetpack Compose
- Surfaces
- Sample Code with Surface
- Material Design Color System scroll all the way down until you get to the “2014 Material Design color palettes”
- Access default icon in SDK
- Material Design Icons
- Material Design Icons Guide
- Add multi-density vector graphics
  - updated

Tutorial Pathways

- Jetpack Compose

Audio

- Base64
  - Encoding with base64
  - Decoding with base64
- Recording and playing audio
  - MediaRecorder
  - Recorder app tutorial
  - Playing audio from byte array
  - Convert file streams to byte arrays and vice versa
  - Fast forward and rewind

Misc

- Restore activity UI state using saved instance state
    - A cool enum Parcelable technique
    - Implementing the Parcelable Interface in Android
    - How can I access all drawables in android?

Appendix: imports

Prepared for EECS 441 by Benjamin Brengman, Ollie Elmgren, Alexander Wu, Wendan Jiang, Yibo Pi, and Sugih Jamin	Last updated: August 25th, 2021