Lab 2: Buttons and State

In case you found something to improve, please tell us!
`https://forms.gle/3U6WrZNyNx2nBXQ38`

This class will teach you how to build an interactive graphical user interface for a tablet application. We will proceed towards building a complete sport tracking application, which you will develop step by step during the upcoming lab sessions.

1 Android Studio Tricks

Here are very useful **Android Studio** tricks you should always use (check *Section 5* of **Lab1** for more detailed explanation on how to use **Android Studio** debug tools):

1. Use **Alt+Enter** (**Option+Enter** for Mac users) when you have an error in your code: put the cursor on the error and click **Alt+Enter**. You can also use it to update the **`gradle`** dependencies to the latest version.
2. Use **Ctrl+Space** to check the documentation of a **`View`**, method or attribute: put the cursor on the object and do **Ctrl+Space**. You can also use it to complete the typing of these objects. Otherwise **Android Studio** always gives a list of suggestions where you can choose the object you need.
3. **ALWAYS CHECK THE COMPILATION ERRORS!** They are usually quite self-explanatory.
4. **ALWAYS DEBUG AND CHECK THE ERRORS IN LOGCAT!** Read the usually self-explanatory errors and click on the underlined blue line to go in the position of the code where the error is.

For more **useful keyboard shortcuts**, please check this LINK[1]

2 Goal

The overview of the graphical user interface (GUI) of the app realized in this lab is shown in Fig. 1. It contains a single screen activity with one **`Image`**, two **`TextField`**s (Username & Password) and two **`Button`**s (”CONFIRM” & ”PICK IMAGE”). The user can select an image from the internal storage of the tablet by pressing the ”PICK IMAGE” button. When the user presses the ”CONFIRM” button, the **`TextField`**s are replaced by **`Text`**s, and ”CONFIRM” & ”PICK IMAGE” buttons are replaced by ”UPDATE” & ”LOG OUT” buttons. The user can update or remove her/his personal information by pressing ”UPDATE” or ”LOG OUT” buttons respectively.

[1] `https://developer.android.com/studio/intro/keyboard-shortcuts`

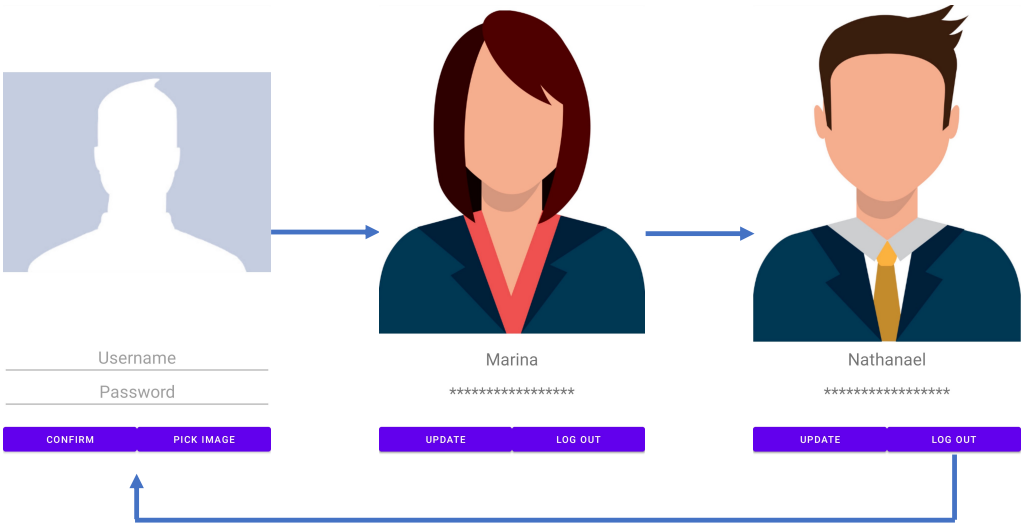

Figure 1: Main Activity Android Layout

To do so you will create a new application running on the tablet. You will design its layout and implement the **MainActivity** Kotlin code which will execute in response to user actions. Let's get started!

2.1 Starting your Android project

Instead of repeating last class' procedure for creating a new Android project, we recommend to use the solution of Lab1 (either the one you coded or the one available on Moodle) to start today's project. We would like **MainActivity** to be lanched everytime the user opens the app. To this end, in the *mobile*'s module, open **AndroidManifest.xml**, and check that **MainActivity** is assigned the **LAUNCHER** category:

```
<activity android:name=".MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

2.2 Initial MainActivity layout

In this lab, we will redesign the GUI of our application, so we will start from a blank slate by deleting the composables in defined in **HomeScreen**. The starting code in **mainacti vity.kt** should look as follows:

```
class MainActivity : ComponentActivity() {

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

        setContent {
            SportsTrackerTheme {
                HomeScreen()
            }
        }
    }
}

@Composable
fun HomeScreen(modifier: Modifier = Modifier) {
    Surface(
        modifier = modifier.fillMaxSize(),
        color = MaterialTheme.colorScheme.background
    ) {

    }
}

@Preview
@Composable
fun HomeScreenPreview() {
    SportsTrackerTheme {
        HomeScreen()
    }
}
```

Now let's add some elements to *HomeScreen()*!

2.3 Add elements to your GUI layout

Let's start by adding a **Column** and an **Image** at the top of the layout. You can download the default user image (**user_image.png**) from Moodle.

```
    Column(
        horizontalAlignment = Alignment.CenterHorizontally,
```

```
) {
    Image(
        painter = painterResource(id = R.drawable.user_image),
        contentDescription = stringResource(R.string.default_user_image)
    )
}
```

Then, add two **TextField**s inside the **Column**. We will see later how these can be replaced by two **Text**s when the user logs in, or or vice versa when the user logs out (see Figure 1).

Each **TextField** is initialized with two parameters: **value** and **onValueChange**. To implement this properly we will need to define a username and a password state variables in the *HomeScreen()* function: **var username by remember { mutableStateOf("") }** and **var password by remember { mutableStateOf("") }**.

Add them to the **TextField**s and implement the **onValueChange** parameter to update the password when the value of the TextField changes. For the pasword, let's also add a **visualTransformation** and set the **keyboardOptions** to **KeyboardType.Password** to hide the password being typed by the user.

```
TextField(
    value = username,
    onValueChange = { newValue ->
        username = newValue
    },
    label = {
        Text(stringResource(R.string.username_hint))
    },
    textStyle = TextStyle(fontSize = 24.sp, textAlign = TextAlign.Center),
    modifier = modifier.fillMaxWidth().padding(bottom = 8.dp)
)
TextField(
    value = password,
    onValueChange = { newValue ->
        password = newValue
    },
    textStyle = TextStyle(fontSize = 24.sp, textAlign = TextAlign.Center),
    label = {
        Text(stringResource(R.string.password_hint))
    },
    visualTransformation = PasswordVisualTransformation(),
```

```
        keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Password),
        modifier = modifier.fillMaxWidth()
    )
```

You might have noticed that the **TextField**s are underlined with a red color. There is an error and to resolve let's click on the TextField and press **ALT+ENTER** and choose the first option **Opt in for ExperimentalMaterial3Api on HomeScreen**.

Let's continue with adding the buttons. They shuld be horizontally placed next to each other, so let's place two **Button**s in a **Row** composable below our **TextField**s.

```
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(top = 16.dp)
    ) {
        Button(
            onClick = { /*TODO*/ },
            modifier = Modifier.weight(1f)
        ) {
            Text(text = stringResource(R.string.confirm_button_text))
        }
        Button(
            onClick = { /*TODO*/ },
            modifier = Modifier.weight(1f)
        ) {
            Text(text = stringResource(R.string.pick_image_button_text))
        }
    }
```

The button have **Modifier.weight(1f)**. Wights are used to set the space occupied by composables in relation to their parent, where each element k will occupy $weight_k / \sum_{i=0}^{N} weight_i$ space in a given direction. In this case, since both buttons have a weight of 1, they will occupy the same space (horizontally): each will be half of the space used by the Row composable containing them.

2.4 State hoisting

Before implementing the other layout with **Text**s, let's implement state hoisting for our current layout. Our application will have two modes:

- Editing mode: When the user can change the username and password, and also pick a picture.
- Displaying mode: When the user can only see the username, password, and picture.

The layout we implemented is for the Editing mode, so let's copy the layout starting from **Column** composable and place it in a new one. Let's call it **HomeContentEditing** and call it inside the **HomeScreen** composable. State hoisting makes the composable stateless, by moving the state to the composable's parent. To do so, need to move **password** and **user name** outside of the **HomeContentEditing** composable, together with the methods that are changing them, and provide them as parameters. Your code should now look as follows:

```
@Composable
fun HomeScreen(modifier: Modifier = Modifier) {
    Surface(
        modifier = modifier.fillMaxSize(),
        color = MaterialTheme.colorScheme.background
    ) {
        var username by remember { mutableStateOf("") }
        var password by remember { mutableStateOf("") }

        HomeContentEditing(
            username = username,
            password = password,
            onUsernameChanged = { newValue -> username = newValue },
            onPasswordChanged = { newValue -> password = newValue },
            modifier = modifier
        )
    }
}

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun HomeContentEditing(
    username: String,
    password: String,
    onUsernameChanged: (String) -> Unit,
    onPasswordChanged: (String) -> Unit,
    modifier: Modifier = Modifier
) {
```

```
        ...
    }
```

Remember that, since now the **onUsernameChanged** and **onPasswordChanged** functions are parameters passed to the **HomeContentEditing** composable, the **onValueChange** method of the username and passwords **TextField**s should be updated accordingly. For example, for the username that would be

```
    TextField(
        value = username,
        onValueChange = onUsernameChanged,
        ...
    )
```

We still are not done. You might notice that our buttons still have **TODO** in their onClick methods, let's declare them as parameters. In **HomeScreen()**:

```
    HomeContentEditing(
            username = username,
            password = password,
            onUsernameChanged = { newValue -> username = newValue },
            onPasswordChanged = { newValue -> password = newValue },
            onContinueButtonClicked = { /* TODO */ },
            onPickImageButtonClicked = { /* TODO */ },
            modifier
        )
```

And, correspondingly, in **HomeScreenEditing** we will list them as parameters:

```
fun HomeContentEditing(
    ...
    onContinueButtonClicked: () -> Unit,
    onPickImageButtonClicked: () -> Unit,
    modifier: Modifier
```

and pass the proper function to the **onClick** methods of the buttons:

```
    Button(
        onClick = onContinueButtonClicked,
        modifier = Modifier.weight(1f)
```

```
    ) {
        Text(text = stringResource(R.string.confirm_button_text))
    }
    Button(
        onClick = onPickImageButtonClicked,
        modifier = Modifier.weight(1f)
    ) {
        Text(text = stringResource(R.string.pick_image_button_text))
    }
```

Before implementing the click listeners for the buttons, let's implement for layout for displaying mode. Copy-paste the layout from the existing composable into a new composable, name it **HomeContentDisplaying**, implement **Text**s instead of **TextField**s and change the buttons accordingly, including the name of the **onClick** methods. In our displaying composable, we won't be able to change the username or the password, so we don't need the **onUsernameChanged** and **onPasswordChanged** parameters. You can create new previews for **HomeContentEditing** and **HomeContentDisplaying** to observe the changes that you make.

```
@Composable
fun HomeContentDisplaying(
    username: String,
    onUpdateButtonClicked: () -> Unit,
    onLogOutButtonClicked: () -> Unit,
    modifier: Modifier = Modifier
) {
    Column(
        horizontalAlignment = Alignment.CenterHorizontally,
    ) {
        Image(
            painter = painterResource(id = R.drawable.user_image),
            contentDescription = stringResource(R.string.default_user_image),
            modifier = modifier
                .fillMaxWidth()
                .height(500.dp)
        )
        Text(
            text = username,
            style = TextStyle(fontSize = 24.sp, textAlign = TextAlign.Center),
            maxLines = 1,
```

```
            modifier = modifier
                .fillMaxWidth()
                .padding(bottom = 8.dp)
        )
        Text(
            text = stringResource(R.string.password_hidden_text),
            style = TextStyle(fontSize = 24.sp, textAlign = TextAlign.Center),
            maxLines = 1,
            modifier = modifier.fillMaxWidth()
        )
        Row(
            modifier = Modifier
                .fillMaxWidth()
                .padding(top = 16.dp)
        ) {
            Button(
                onClick = onUpdateButtonClicked,
                modifier = Modifier.weight(1f)
            ) {
                Text(text = stringResource(R.string.update_button_text))
            }
            Button(
                onClick = onLogOutButtonClicked,
                modifier = Modifier.weight(1f)
            ) {
                Text(text = stringResource(R.string.log_out_button_text))
            }
        }
    }
}

@Preview
@Composable
fun HomeScreenDisplayingPreview() {
    SportsTrackerTheme {
        HomeContentDisplaying("username", {}, {})
    }

}
```

```
@Preview
@Composable
fun HomeScreenEditingPreview() {
    SportsTrackerTheme {
        HomeContentEditing("", "", {}, {}, {}, {})
    }
}
```

3 Reacting to clicks on Buttons

3.1 The Confirm button

When we click on the ”Confirm” button, we are expecting the change the mode, from Editing to Displaying, i.e. the **TextField**s to be removed and replaced with **Text**s and the text of the **Button**s to be updated. We can do that from the top-level composable (**HomeScreen**), using a new state variable called **isEditingMode** initialized to **true**, since we want our application to start in Editing mode. When we click on the ”Confirm” button, we just need to change the state to false. To implement to changing of screens we need and if-else block where the variable **isEditingMode** is the condition. If **isEditingMode**is true, we want to call the **HomeContentEditing** composable, if it is false we want to call the **HomeContentDisplaying**. Run the application and test the changing of modes.

```
    var isEditingMode by remember { mutableStateOf(true) }

    if (isEditingMode) {
        HomeContentEditing(
            username = username,
            password = password,
            onUsernameChanged = { newValue -> username = newValue },
            onPasswordChanged = { newValue -> password = newValue },
            onContinueButtonClicked = { isEditingMode = false },
            onPickImageButtonClicked = { /* TODO */ },
            modifier
        )
    } else {
        HomeContentDisplaying(
            username = username,
            onUpdateButtonClicked = { /*TODO*/ },
```

```
            onLogOutButtonClicked = { /*TODO*/ },
            modifier)
    }
```

Indeed, when "Confirm" is pressed, username and password are no longer editable, so **TextField**s are substituted by the corresponding **Text**s.

3.2 The "PickImage" button

Now let's define **onPickImageButtonClicked** function. To perform this task, we have to request data from outside of the app (from an image chooser).

External apps are invoked via Intents. In our case, the action must be "ACTION GET CONTENT" and the type "image/*". The Intent is then launched, so that the (external) image chooser app can come in the foreground. We also need to add another state variable to remember the Uri (location) of the image that was received from the intent. We need to add a type **Uri?** to the MutableState, and initialize it to **null**.

```
    onPickImageButtonClicked = {
        val intent = Intent(Intent.ACTION_GET_CONTENT)
        intent.type = "image/*"
         resultLauncher.launch(intent)
    },
```

The launcher callback is defined inside the **HomeScreen** composable:

```
    var imageUri by remember { mutableStateOf<Uri?>(null) }
    var resultLauncher = rememberLauncherForActivityResult(
        contract = ActivityResultContracts.StartActivityForResult(),
        onResult = { result ->
            if (result.resultCode == Activity.RESULT_OK) {
                val uri = result.data?.data
                imageUri = uri
            }
        }
    )
```

In order to see the image that we picked, we will need to modify our **HomeContentEdit**⌋**ing** and **HomeContentDisplaying** composables. Let's first add a Uri parameter as a first pameters, **imageUri: Uri?**. In both cases, we should pass the **ImageURI** (that we retrieved

from the resultLauncher) as an argument when calling them from **HomeScreen** as a first argument when calling

Loading image from files or from the internet is a ”heavy” process that should be done asynchronously in the background (not blocking the UI). To do this we will use an external library that that. Add **`implementation("io.coil-kt:coil-compose:2.4.0")`** in the **`build.gradle.kts`** in the **`mobile`** module. Click on **`Sync now`** and wait for it to finish.

When it is finished we can use the **`AsyncImage`** composable. We want to show the current **`R.drawable.user-image`** when **`imageUri`** is null, and when we Uri is not null (the user as not selected any image). Instead, we want to display the Image from the **`imageUri`**. This can be achieved by adding the following code both in the **HomeContentEditing** and in the **HomeContentDisplaying** composable. Note that this solution present redundant code. A more elegant (but a bit more complex) aldernative would have been to use a dedicated composable function for displaying the image, and reusing it in both the editing and displaying interfaces.

```
if (imageUri == null) {
    Image(
        painter = painterResource(id = R.drawable.user_image),
        contentDescription = stringResource(R.string.default_user_image),
        modifier = modifier
            .fillMaxWidth()
            .height(500.dp)
    )
} else {
    AsyncImage(
        model = imageUri,
        contentDescription = stringResource(R.string.picked_user_image),
        modifier = modifier
            .fillMaxWidth()
            .height(500.dp)
    )
}
```

3.3 ”Update” and ”Logout” buttons

Using the same strategy illustrated for the ”Confirm” button, implement the **`onUpdateButtonClicked`** and **`onLogOutButtonClicked`**. Their functionality should be as follows:

- Tapping on the ”Log out” button should remove the user information from the GUI,

returning it to its initial state: usernames and passwords are blank, and the default image is displayed and we should return to Editing mode.

- The ”Update” button should instead allow the user to change username, password, and picture without resetting it, i.e. changing to ‘Editing mode.

Both click listeners must properly manipulate the modes, and Buttons and the content of texts and images.