Lab 4: Interfacing Android Wear devices

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

In this lab, we will employ the WearAPI to synchronise data across devices. In particular, we will code a distributed application that, upon user login, will send the username and profile from an Android tablet to an Android smartwatch.

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 Sending Login credentials to the wearable device

The overview of the intended app functionality on tablet and watch is described in Fig. 1. As in the previous lab, on the tablet the user can select his profile image and enter his username and password. In the LoginProfile screen, when the user presses the ”ENTER” button, the tablet app sends the image and username to the watch, which displays it. To this end, the tablet and watch should be paired via Bluetooth. This can be done via the WearOS app on your tablet/smartphone. If not present on your device, this can be installed from the googlePlay store. Also keep into account that the wear and tablet modules should be compiled on the same computer for the connection to work.

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

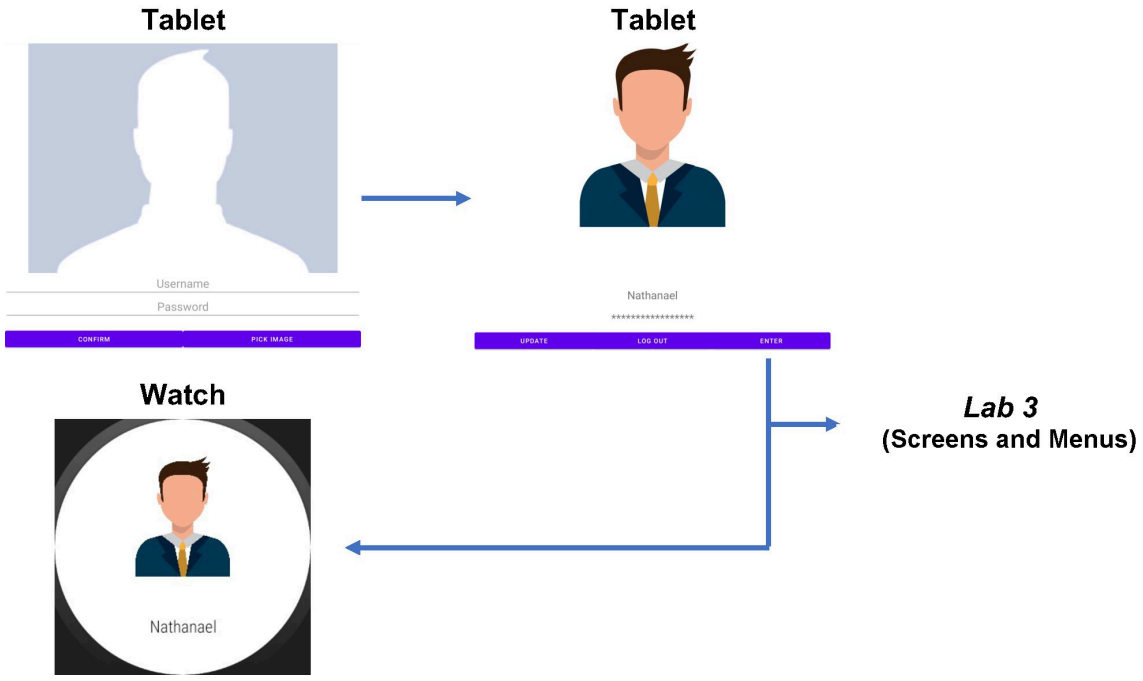

Figure 1: App structure for lab 4

3 Communicating between tablet and watch

Let's continue building our sport tracking app, from the structure described in *Lab 3*, adding some more features. Open the *Lab 3* project, which you can take from Moodle. Alternatively, you can use your own implementation.

In this lab, we will use the Data API to establish a communication between watch and tablet. To this end, first we need to register the service in the *mobile*'s and *wear*'s **build.gradle** files, adding the following lines to both:

```
dependencies {
    ...
    implementation("com.google.android.gms:play-services-wearable:18.1.0")
    ...
}
```

In the communication we want to implement between tablet and watch there will be two sides: the 'sending side' and the 'receiving side'. On the sending side (the tablet in this lab), a **dataClient** object is used to interface with the Data API. On the receiving side (the watch in this lab), **OnDataChangedListener()** callbacks are executed when receiving data.

4 Using the Wear Data API

4.1 Mobile module: `MainActivity` class

We want to send the profile data from the LoginProfile screen. Let's start with the declaration and initialization of the **`dataClient`** object inside the Kotlin Activity class, but outside any method. We declare it as *lateinit* since the object will be initialized in the *onCreate()* method. The dataClient will allow us to send items using the Wear Data API. In particular, we will use the 'dataClient.putDataItem' method in a few steps.

```
private lateinit var dataClient: DataClient

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

    dataClient = Wearable.getDataClient(this)

    setContent {
    ...
    }
}
```

Since we want to send the data once the user presses the "ENTER" button, we call a custom function (**`sendDataToWear()`**) inside the onEnterButtonClicked:

```
composable("login") {
    LoginProfileScreen(
        onEnterButtonClicked = {
            shouldShowBottomMenu = true

            sendDataToWear(username, imageUri)
            ...
        }
    )
}
```

You may notice that username and imageUri are unavailable, and we will need to receive them from the LoginProfileScreen. Let's create a data class in a new data class file called **`LoginInfo`** that will store the login information, the username, and the imageUri.

```
data class LoginInfo(val username: String, val imageUri: Uri?)
```

Now we will have to modify the signature of the **onEnterButtonClicked** callback method, because will have to pass a LoginInfo object when we call this callback in the onClick method of our "Enter" button. The parameters of **LoginProfileContentDisplay** and **LoginProfileScreen** should be modified accordingly.

```
fun LoginProfileContentDisplay(
    ...
    onEnterButtonClicked: ((LoginInfo) -> Unit),
    modifier: Modifier = Modifier
) {
    ...
    Button(
        onClick = {
            val loginInfo = LoginInfo(username, imageUri)
            onEnterButtonClicked(loginInfo)
        },
        modifier = Modifier.weight(1f)
    ) {
        Text(text = stringResource(R.string.enter_button_text))
    }
    ...
}
```

Now we can go back to the MainActivity class, where we can use the login info that were passed from the LoginProfileScreen.

```
composable("login") {
    LoginProfileScreen(
        onEnterButtonClicked = { loginInfo ->
            shouldShowBottomMenu = true

            val username = loginInfo.username
            val imageUri = loginInfo.imageUri

            sendDataToWear(username, imageUri)
            ...
        }
    )
}
```

We are now ready to implement the **sendDataToWear(username: String, imageUri:**

Uri?) function, as a private function inside the **MainActivity** class. In it, the data items (image and username) are embedded in a **PutDataMapRequest** object, and in particular in its dataMap (i.e. payload), and then send using the DataAPI.

As a first step, we manipulate the image to the proper size and format, using some methods from the **Mediastore**, **Bitmap**, and **Matrix** classes. The complete documentation of these classes is available on the Android developer website.

We grab the image content from its URI (point 1 in the code below). Then, to minimize the communication bandwidth between the tablet and watch, we scale it (point 2 in the code) using **createScaledBitmap(...)**. We also rotate the image (3) to the proper orientation to be displayed on the watch. Finally, convert it from the Bitmap to the ByteArray format (4).

```
private fun sendDataToWear(username: String, imageUri: Uri?)
{
    //1
    var imageBitmap = MediaStore.Images.Media.getBitmap(
                          this.contentResolver, imageUri)

    //2
    var ratio:Float = 13F

    val imageBitmapScaled = Bitmap.createScaledBitmap(imageBitmap,
                              (imageBitmap.width / ratio).toInt(),
                              (imageBitmap.height / ratio).toInt(),
                              false)

    //3
    val matrix = Matrix()

    imageBitmap = Bitmap.createBitmap(imageBitmapScaled, 0, 0,
    (imageBitmap.width / ratio).toInt(),
    (imageBitmap.height / ratio).toInt(), matrix, true)

    //4
    val stream = ByteArrayOutputStream()
    imageBitmap.compress(Bitmap.CompressFormat.PNG, 100, stream)
    val imageByteArray = stream.toByteArray()
    ...
}
```

The created imageByteArray can then be used as a field in a *dataMap* object (see code, below). Each **dataMap** field is defined as a key/value pair, e.g. "profileImage"/*imageByteArray*.

The dataMap is then encapsulated in an object of type **PutDataRequest**, which must also specify the path at which the data must be sent to the connected device (e.g.: "/userInfo").

Finally, the **PutDataRequest** object (named **request** in the code below), can be sent using the Wear API with the data client we initialized at the beginning of this Section. Before that, we set the request as "urgent" to ask for its delivery as soon as possible. Otherwise, the OS may decide to delay transmission until enough data is collected to save energy.

```
private fun sendDataToWear(username: String, imageUri: Uri?)
{
    ...
    val request: PutDataRequest = PutDataMapRequest.create("/userInfo").run {
        dataMap.putByteArray("profileImage", imageByteArray)
        dataMap.putString("username", username)
        asPutDataRequest()
    }

    request.setUrgent()
    val putTask: Task<DataItem> = dataClient.putDataItem(request)
}
```

As you can see, we defined "/userInfo" as the path, and "profileImage" and "username" as the labels for the username and image respectively. This information will be used on the receiving side i.e. the wear module.

As a last touch, as part of the NavHost in **MainActivity** we can chech in that neither **userName** nor **imageUri** are **""** / **null**, when navigating from **LoginProfileScreen** to **NewRecordingScreen**. Otherwise, we may show Toast[2] instead of navigating to the **NewRecordingScreen** and sending the data to Wear device:

```
val context = LocalContext.current
...
Toast.makeText(context, "Pick an image and a username!",
               Toast.LENGTH_SHORT).show()
```

[2] https://developer.android.com/guide/topics/ui/notifiers/toasts

4.2 Wear module: the `MainActivity` class

This section focuses on the receiving side of the Data connection: the Wear module. Here, we want to listen, in the Wear activity, for data events. Keep in mind that in different communication patterns, such as streaming heart rate data from watch to tablet, the role of sender and receiver could be reversed.

In the Wear module, we first override the **`onResume()`** and **`onPause()`** methods in the Activity class, which are called every time the activity goes in the foreground or is pushed to the background, respectively.

Inside those methods, we call **`Wearable.getDataClient(this).addListener(this)`** and **`Wearable.getDataClient(this).removeListener(this)`**. In this way, we notify that our activity is interested in listening for data layer events when in the foreground.

```
override fun onResume() {
    super.onResume()
    Wearable.getDataClient(this).addListener(this)
}

override fun onPause() {
    super.onPause()
    Wearable.getDataClient(this).removeListener(this)
}
```

To be notified of data changes, the wear app must override the **`onDataChanged`** callback, available by implementing the **`DataClient.OnDataChangedListener`** interface in its Activity class definition. We need two class-level variables, username and bitmap, so that we can pass them to our HomeScreen composable in the onCreate method. We also need to update them appropriately in the onDataChanged callback, so that we can see the latest changes on the UI.

```
class MainActivity : ComponentActivity(), DataClient.OnDataChangedListener {
    private var bitmap by mutableStateOf<Bitmap?>(null)
    private var username by mutableStateOf("")
    ...
}
```

Whenever a data item object is created, deleted, or changed, the system triggers the **`onDataChanged()`** callback on all connected nodes. Inside this function, we look at pending *dataEvents*, check if the path and key match, and retrieve the data.

Notice that there may be multiple data *event*s notified by the APIs. Therefore, we cycle through them and get, from each event dataMap, the payload corresponding to the dataMap fields.

Once the payload is extracted, it can be converted to the appropriate format (e.g. from ByteArray to Bitmap in the case of the profile picture). More information on *.filter* and scope functions in Kotlin (including *.let*) is available here[3] and here[4].

```
override fun onDataChanged(dataEvents: DataEventBuffer) {
    dataEvents.filter { it.type == DataEvent.TYPE_CHANGED &&
                        it.dataItem.uri.path == "/userInfo" }
        .forEach { event ->
            val receivedImageBytes: ByteArray? =
                DataMapItem.fromDataItem(event.dataItem).dataMap.
                            getByteArray("profileImage")

            receivedImageBytes?.let {
                bitmap = BitmapFactory.decodeByteArray(
                            receivedImageBytes, 0,
                            receivedImageBytes.size)
            }

            username  = DataMapItem.fromDataItem(event.dataItem)
                        .dataMap.getString("username") ?: ""
        }
}
```

To show the received items on the smartwatch, we set the bitmap to the received image and set the value of Text to the received username. To do this, we need to update the **HomeScreen** composable, adding username and bitmap as parameters. We also need to modify the logic for displaying the bitmap, as it should show a default picture if nothing has been received yet, otherwise the received image.

```
@Composable
fun HomeScreen(username: String, bitmap: Bitmap?,
               modifier: Modifier = Modifier) {

    ConstraintLayout(modifier = modifier.fillMaxSize())
                                        .background(color = Color.Black)) {
```

[3] https://kotlinlang.org/docs/collection-filtering.html
[4] https://kotlinlang.org/docs/scope-functions.html

```
        val (image, text) = createRefs()

        val context = LocalContext.current

        val displayBitmap = if (bitmap == null) {
            ImageBitmap.Companion.imageResource(
                        context.resources, R.drawable.ic_logo)
        } else {
            bitmap.asImageBitmap()
        }

       Image(
            bitmap = displayBitmap,
        ...

    }
}
```

The dislayBitmap variable assignment can be refactored to follow the Kotlin syntax more closely with the following code:

```
val displayBitmap = bitmap?.asImageBitmap()
    ?: ImageBitmap.Companion.imageResource(context.resources, R.drawable.ic_logo)
```

Now that we can finally show the profile picture and username on the watch!

5 Navigation arguments

In addition to sending the login credentials to the watch, we would also like to share them among Screens on the tablet, in particular from **LoginProfileScreen** to the **NewRecordingScreen**.

5.1 Send username to the NewRecordingScreen

As you may recall from the lecture, data can be passed between screens as part of a route. So, first, inside **NavHost** in the MainActivity we will have to modify the composable with route ”newRecording” to ”newRecording/username” and we retrieve the username from the backStackEntry object. Now that we have the username, we can pass it to **NewRecordingScreen** as a parameter, and update the Welcome text appropriately.

```
composable("newRecording/{username}") { backStackEntry ->
    val username = backStackEntry.arguments?.getString("username") ?: ""

    NewRecordingScreen(
       username,
       onLogoutClicked{...}
    )
}
```

The only missing part is passing the username when we navigate to the **NewRecordingScreen** from the **LoginProfileScreen**. So inside the **onEnterButtonClicked** callback pass the username as part of the route.

```
composable("login") {
    LoginProfileScreen(
        onEnterButtonClicked = { loginInfo ->
            ...
            navController.navigate("newRecording/$username")
            ...
```

5.2 Send the imageUri to the NewRecordingScreen

We would also need to pass the imageUri as part of the route, but if you recall from the lecture, we can also pass a primitive type as a navigation parameter, and imageUri is of type **Uri**, which is not a primitive type. For this reason, we will need to encode the imageUri as a String and pass it in the route.

```
composable("login") {
    LoginProfileScreen(
        onEnterButtonClicked = { loginInfo ->
            ...
            val imageUri = loginInfo.imageUri

            val uriString = URLEncoder.encode(
                imageUri.toString(),
                StandardCharsets.UTF_8.toString()
            )

            navController.navigate("newRecording/$username/${uriString}") {
            ...
```

We need to receive the imageUri as a String in the composable of the NavHost of **newRecording** we need to parse the String back to a Uri object and we can do that with **Uri.parse(imageUriString)** and pass it as a parameter to the **NewRecordingScreen**.

```
composable("newRecording/{username}/{imageUriString}") { backStackEntry ->
    val username = backStackEntry.arguments?.getString("username") ?: ""

    val imageUriString =
        backStackEntry.arguments?.getString("imageUriString")
    val uri = if (!imageUriString.isNullOrEmpty()) {
        Uri.parse(imageUriString)
    } else {
        null
    }

    NewRecordingScreen(
        username,
        uri,
        ...
    )
}
```

Inside the **NewRecordingScreen** we will need to do a similar logic as we did in **LoginProfileScreen**. We should the default initial image when the passed **imageUri** is null, and when it is not null, we show the image from the imageUri.

```
    ...
    if (imageUri == null) {
        Image(
            painter = painterResource(id = R.drawable.user_image),
            ...
        )
    } else {
        AsyncImage(
            model = imageUri,
            ...
        )
    }
```

Run the application and check if the image and the username are passed correctly. If that works, try navigating on the screen in the bottom navigation bar.

Our application crashed because we didn't modify the code for navigation to the **newRecordingScreen** from the bottom navigation bar. We currently have access to the **username** and the **imageUri** inside the **onEnterButtonClicked** callback. To change that we need to introduce activity-level variables for the username, the imageUri, and the uriString.

```
class MainActivity : ComponentActivity() {

    private lateinit var dataClient: DataClient
    private var username by mutableStateOf("")
    private var imageUri by mutableStateOf<Uri?>(null)
    private var uriString by mutableStateOf("")
    ...

}
```

Inside the **onEnterButtonClicked** callback we just need to remove the *val* keywords for username, imageUri and uriString, so that we can access the activity-level variables we just defined. Now let's go and fix the **NavigationBarItem** that displays the **NewRecordingScreen**. First, we need to modify the *selected* parameter, since the route is no longer "newRecording", but instead newRecording/$username/$uriString. Hence, we will set it to "false" if it does not match the beginning of the route name. The final step is to assign the correct route in the onClick block, which has to he the same as in the onEnterButtonClicked callback.

```
    NavigationBarItem(
        selected = currentRoute?.startsWith("newRecording") ?: false,
        onClick = {
            navController.navigate("newRecording/$username/$uriString")
        },
        ...
```

.