Lab 3: Activities and Intents

In case you found something to improve, please tell us!
`https://forms.gle/Nf27cXFf7AwaBL55A`

This class teaches you how to use different activities to manage a profile. In particular, you will learn how to use intents to send actions between activities.

1 Android Studio Tricks

Here are very useful **Android Studio** tricks you should always use (check *Section 5* of **Lab1b** for more detailed explanation on how to use **Android Studio** 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 Introduction to Activities and Intents

Activities are components which consist of a layout and can do different things such as handling user interactions, starting other activities and have a life-cycle. Most of the time, an **`Activity`** is a *screen*: when the user goes from screen to screen, it is translated to moving from one **`Activity`** to another.

This lab shows the interaction between three activities. To start a new activity from the current one, an **`Intent`** is required: it is a message that requests an action from other components through the Android system. Additionally, an app can register for specific events (common intents are documented[2]) by using an **`intent-filter`**, which is defined in

[1] <https://developer.android.com/studio/intro/keyboard-shortcuts>
[2] <https://developer.android.com/guide/components/intents-common>

AndroidManifest.xml. When creating a new project, the manifest declares the existence of one activity, with the following **intent-filter**:

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

In this case, the **intent-filter** specifies that the activity is the main one, that is to say that it is the one which is started when the user taps on the app icon from the list of apps.

3 Lab 3 main structure

Let's continue building our sport tracking app that we started in *Lab 2*, by adding some more features. For example, we want to create a registration form for new users to enable them to login. For this, we need 3 activities: **LoginActivity**, whose layout was created in *Lab 2*, **EditProfileActivity**, which will show the registration form, and **MainActivity**, which is the home screen of the app after the user logs in. Moreover, we will have a **Profile** class (not an activity), which will contain the main elements of the user profile and will be sent through different activities, using intents, and the wear app. The main structure is described in Figure 1.

Open the *Lab 2* project, which you can take from Moodle (*Solution Lab 2 - 2020), or you can use the project you implemented yourselves following *Lab 2*.

4 Mobile module

4.1 The Profile class

Let's start with creating a **Profile** class in the *mobile* module. We choose to have five main elements for the user profile: username, password, height, weight and path of the user photo (feel free to add some more, like a birth date picker). We add these elements as class fields and we add also a constructor with username and password.

```
public class Profile implements Serializable{
    private static final String TAG = "Profile";

    protected String username;
```

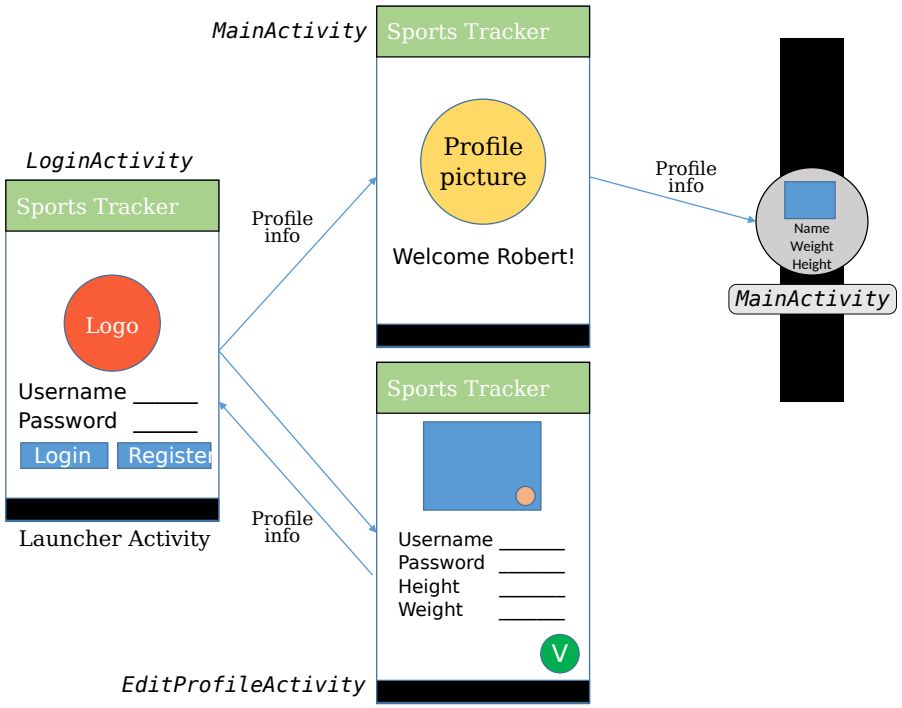

Figure 1: Sport tracker app structure for lab 3

```
    protected String password;
    protected int height_cm;
    protected float weight_kg;
    protected String photoPath;

    public Profile(String username, String password) {
        this.username = username;
        this.password = password;
    }
}
```

Note that the fields in **Profile** have a visibility **protected**, which means they can be accessed from any class within the same package. Also, the **Profile** class needs to implement the **Serializable** interface to send different type of data together (not one by one) through an **Intent**. The process of serialization consists of converting the **Profile** class into a byte stream and further reconstructing it at the receiver part.

4.2 The LoginActivity

4.2.1 Check user registration (fake)

Now we can use the **Profile** class to *fake* the check of a registered user. When we tap *Login*, we want to see an error message saying that the user is not yet logged if we didn't complete the registration form. First, we need to create a private **Profile** field that we can call **userProfile**, visible to **LoginActivity**, and we set to null.

```
private Profile userProfile = null;
```

Then, we modify the **onClick(...)** callback of the *Login* button by checking if the **userProfile** is **null**. If it is **null**, we show a red message in the already present **Textview** saying that the user is not registered (Figure 2). Later when the user will be registered, once pressed *Login*, we will start the **MainActivity**. For now, **userProfile** will always be null, since we haven't yet implemented the user registration form.

```
if(userProfile != null) {
    // TODO: start the MainActivity once created (in little time :) )
} else {
    TextView textView = findViewById(R.id.LoginMessage);
    textView.setText("You are not registered yet!");
    textView.setTextColor(Color.RED);
}
```

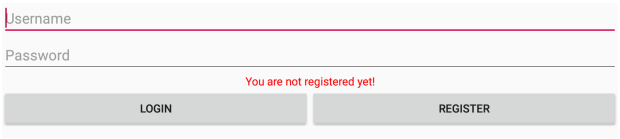

Figure 2: Error message for user not registered

4.3 The EditProfileActivity

4.3.1 Creating the layout

The first step to register an account in our sport tracker is to design a profile input form. We will call it **EditProfileActivity**, since we will reuse the same class both for registration and for later editing the user profile once logged in. Remember to always add your

activity in the **AndroidManifest.xml**, although **Android Studio** should do that automatically. If Android Studio detects an activity which is not in the manifest, it will offer to do it for you.

Based on the knowledge acquired in *Lab 2*, create an adequate layout for the **EditProfileActivity**, which will show a form to fill-in. The appearance does not matter (for this lab!), we just require the following different elements: some text fields for *Username*, *Password*, *Height* and *Weight*, an image container for the profile picture and two buttons: one for choosing a picture and one for saving the form. Figure 3 shows an example of a valid layout. This one has been created with a **ConstraintLayout**: feel free to experiment

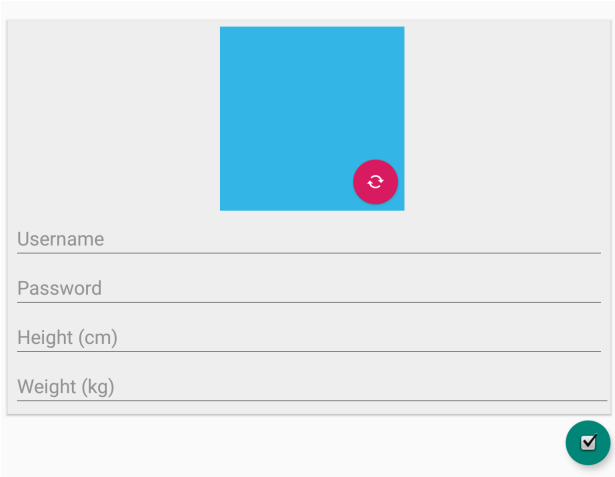

Figure 3: Edit profile layout

something else than the **LinearLayout** we began with. Be creative! Just remember to give an id to each component so that you will be able to program it from the activity side. The ids we are using are:

- **userImage** for the profile picture
- **chooseImageButton** with callback **chooseImage** to select a profile picture
- **editUsername** to set the username
- **editPassword** to set the password
- **editHeight** to set the height
- **editWeight** to set the weight
- **editUserButton** with callback **editUser** to save the form

4.3.2 Starting activity for a result

We need to start **EditProfileActivity** by tapping the *register* button of the **LoginActivity**. We also want to get the profile information back to the **LoginActivity** once we

have registered it. In order to do this we need to use a method called **startActivityForResult(...)** which takes as input an **Intent** object and an integer request code as identifier of the result request. The latter is a constant field with arbitrary value, that we call **REGISTER_PROFILE**. First, we create this field in the **LoginActivity**.

```
private static final int REGISTER_PROFILE = 1;
```

Then, in the callback of the *register* button, we call the method with an Intent between **LoginActivity** and **EditProfileActivity**.

```
Intent intentEditProfile =
        new Intent(LoginActivity.this, EditProfileActivity.class);
startActivityForResult(intentEditProfile, REGISTER_PROFILE);
```

Now if we run the app and tap the register button, we will see the layout of our **EditProfileActivity** without any functionality. In the next sections we will explain how to choose a profile picture and how to send back the result of our profile to the **LoginActivity**.

4.3.3 Choosing a profile picture

Let us focus on the profile picture. The **chooseImageButton** is used to request the action of opening the storage from where we will choose our picture. As we did in Lab 2, we need either a java or an XML callback. In this case, we choose an XML callback through the **onClick** which will create a method **chooseImage(...)**.

Since we need to request an action, we call a new **Intent**. We need to set which type of data the intent will return, in this case any type of image. Then, we set the action which will open the storage to get a particular content. We use again **startActivityForResult(...)**. The activity that responds must be designed to return a result and it will have the same identifying code. It means that an activity can start several other activities each with a different code. That way, when receiving the result, it is easy to identify which result comes from which other activity. In our case, we define the **PICK_IMAGE** constant so it's easier to understand what the code is doing.

The implementation of **chooseImage(...)** in **EditProfileActivity** is the following:

```
// Usually at the top of the class
private static final int PICK_IMAGE = 1;

// Anywhere with the other methods of the class
public void chooseImage(View view) {
```

```
    Intent intent = new Intent();
    intent.setType("image/*");
    intent.setAction(Intent.ACTION_GET_CONTENT);
    startActivityForResult(
        Intent.createChooser(intent, "Select Picture"), PICK_IMAGE);
}
```

In this activity, we need to receive a result from the picture we choose. The action we want to perform is given by the **createChooser(...)**. This method allows to send the user to a different app with the capability **ACTION_GET_CONTENT** of type **image**/*. Since there can be different apps that respond to this request, this is defined as chooser. The user can choose which app to start. In the case where there is only one app, it directly opens it. The string argument is assigned to the title that will be shown to the user as a chooser dialog.

If you try the app now, tapping the button to choose an image should let you select an image file. In this case, though, we are not yet setting the image in the **ImageView** of **EditProfileActivity** once we choose the image. In order to do this, we need to implement the **onActivityResult(...)** in **EditProfileActivity**. We need the request code **PICK_IMAGE** and the default result code for validated image which is **RESULT_OK**, given by the **AppCompatActivity**. In the method, we want to set the image chosen in the **ImageView** that we called **userImage**. To add the method **onActivityResult(...)** you can tap *Ctrl+O*, which is a shortcut to override methods of the current class and type the name of the method. Then, you can implement the following code (first, declare a **private** attribute of type **File** in **EditProfileActivity** called **imageFile** where we will store the profile image):

```
if (requestCode == PICK_IMAGE && resultCode == RESULT_OK) {
    Uri imageUri = data.getData();
    imageFile = new File(getExternalFilesDir(null), "profileImage");
    try {
        copyImage(imageUri, imageFile);
    } catch (IOException e) {
        e.printStackTrace();
    }
    final InputStream imageStream;
    try {
        imageStream = getContentResolver()
                .openInputStream(Uri.fromFile(imageFile));
        final Bitmap selectedImage = BitmapFactory.decodeStream(imageStream);
        ImageView imageView = findViewById(R.id.userImage);
        imageView.setImageBitmap(selectedImage);
```

```
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}
```

In the code above, we use a **Uri** (Uniform Resource Identifier reference) object, which represents the image path we get from the intent. Since the **Uri** object we get after choosing the picture is a temporary path that we cannot reuse later, we need to save (or better *copy*) the image into a **File** (the attribute **imageFile** we created before). For this, we need to implement the private method **copyImage(...)** in the **EditProfileActivity**. First, we open a **FileInputStream** to read the image file considering the **Uri** object, then we save it to a new **File** by using a **FileOutputStream** (code below).

```
private void copyImage(Uri uriInput, File fileOutput) throws IOException {
    InputStream in = null;
    OutputStream out = null;

    try {
        in = getContentResolver().openInputStream(uriInput);
        out = new FileOutputStream(fileOutput);
        // Transfer bytes from in to out
        byte[] buf = new byte[1024];
        int len;
        while ((len = in.read(buf)) > 0) {
            out.write(buf, 0, len);
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        in.close();
        out.close();
    }
}
```

After copying the image into a **File**, we create a Bitmap to set the **ImageView**. You can try to run the app now and you will see the image that you chose in the view.

4.3.4 Process the form

After choosing the image, we need to fill the form and send it back to the **LoginActivity**. We will do this in the callback of the **editUserButton**. First, we need to get the username and password from the first two **TextView**s and create a new **Profile** with these two fields.

```
TextView username = findViewById(R.id.editUsername);
TextView password = findViewById(R.id.editPassword);
Profile userProfile = new Profile(username.getText().toString(),
            password.getText().toString());
```

Then, we can set all the other fields of **newUser** as done in the following code. The result is shown in Figure 4.

```
TextView height = findViewById(R.id.editHeight);
TextView weight = findViewById(R.id.editWeight);
try {
    userProfile.height_cm = Integer.valueOf(height.getText().toString());
} catch (NumberFormatException e) {
    userProfile.height_cm = 0;
}
try{
    userProfile.weight_kg = Float.valueOf(weight.getText().toString());
}catch (NumberFormatException e){
    userProfile.weight_kg = 0;
}
if (imageFile == null) {
    userProfile.photoPath = "";
} else {
    userProfile.photoPath = imageFile.getPath();
}
```

4.3.5 Send back result to LoginActivity

Once the form has been processed, we need to give a result to the **startActivityForResult(...)** we called from **LoginActivity** to the **EditProfileActivity**. We need to create a new **Intent** to pass our **userProfile**. Then, we need to set the result code to the default **RESULT_OK** and finish the activity. To do this last step call the method **finish()**, which removes the activity from the navigation stack.

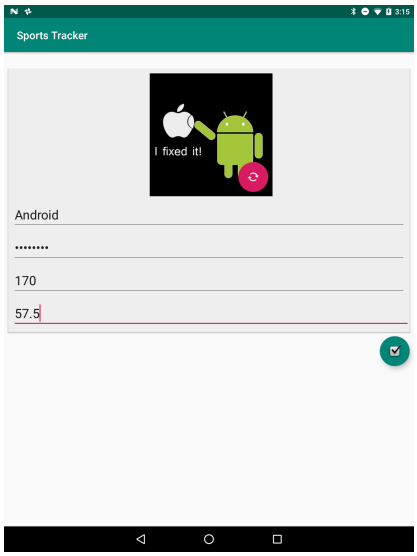

Figure 4: Profile form filled

```
Intent intent = new Intent(EditProfileActivity.this, LoginActivity.class);
intent.putExtra("userProfile", userProfile);
setResult(AppCompatActivity.RESULT_OK, intent);
finish();
```

Finally, the last step is to override the method **onActivityResult(...)** in the **LoginActivity** to receive the user profile and fill the username and password automatically. Use the same shortcut mentioned before to override a method. Then, we write the following code in the method.

```
if (requestCode == REGISTER_PROFILE && resultCode == RESULT_OK) {
    userProfile = (Profile) data.getSerializableExtra("userProfile");
    if (userProfile != null) {
        TextView username = findViewById(R.id.Username);
        username.setText(userProfile.username);
        TextView password = findViewById(R.id.Password);
        password.setText(userProfile.password);
    }
}
```

If we run the app now, we will be able to go back to **LoginActivity** after filling the form and see username and password automatically set, as shown in Figure 5.

To make the app more robust, it is important to check that the user has properly filled the required fields of the registration form before sending them back to **LoginActivity**.

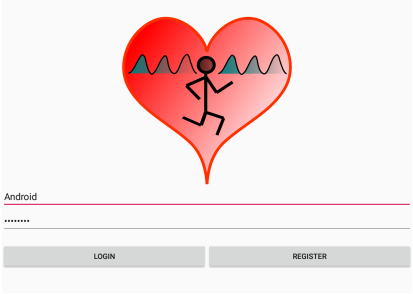

Figure 5: Automatic setting username and password after registration

For example, you can check that the username and password are not empty. You can also, display an error message if the form was not properly completed. This can be done in the callback function of **`editUser`** button **`View`**. Try to do this as an exercise of what you have previously learned!

In the next section we will learn how to send the user profile to the smart watch, once logged in and visualized the home screen.

4.4 The `MainActivity`

The **`MainActivity`** is traditionally the activity the user will be interacting the most with. Think of it as the home view (once the user is logged-in if it applies). A calendar app will display events registered, a camera app will display the live-view of the camera sensor, a web-browser will display the browsing interface, a social network app will display the news feed... In our case, the main content will be coming later in the next weeks. In the meantime, we will have a very simple place-holder layout with minimal content.

Start by creating the new activity from Android Studio's menus: **File → New → Activity → Empty activity**. Fill-in the layout to have at least an **`ImageView`** and a **`TextView`**: we will greet our user with the profile picture and a custom text. Try to send the profile picture and the username to do this, by using the method **`startActivity(...)`** with an **`Intent`** in input and getting the **`Intent`** in the **`onCreate(...)`** method of the **`MainActivity`** by using **`getIntent()`**. Note that you have to create a new **`FileInputStream`** and pass the photoPath to it to create the **`Bitmap`** for your **`ImageView`** as we did while choosing the picture.

Before going on to the Wear communication part, don't forget to link the *Login* button with starting this brand new **`MainActivity`**!

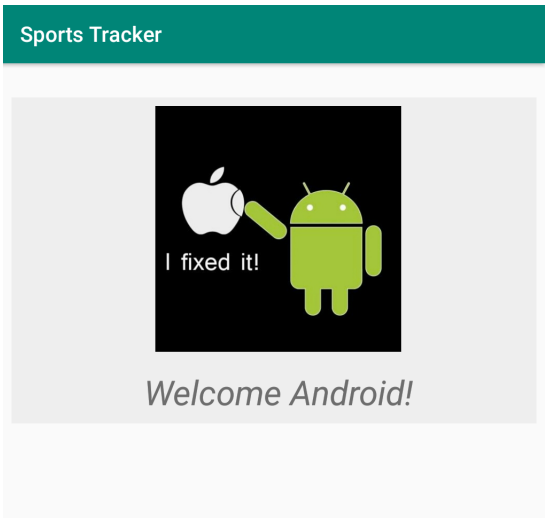

Figure 6: Welcome screen of our sport tracker app

5 Interfacing with Android Wear

5.1 Adding the `WearService` service

Add the **`WearService.java`** file found on Moodle to **both** modules (*mobile* and *wear*) with a drag-and-drop in Android Studio's project tree. It will update the package name of the file provided automatically. If it does not, simply apply the fix it will suggest.

This newly added service is a class we created to make the communication between the mobile and the watch easier, by providing a well tested solution.

This **`WearService`** is relying on constants generated at build time to prevent typing mistakes. We therefore need to perform the required modifications to the *project*'s **`build.gradle`** file, which is defining build-time project-wide string constants:

```
allprojects {
    repositories {
        ...
    }

    // Constants defined for all modules, to avoid typing mistakes
    // We use it for communication using the Wear API
    // It is a key-value mapping, auto-prefixed with "W_" for convenience
    project.ext {
        constants = [
                path_start_activity    : "/START_ACTIVITY",
```

```
                path_acknowledge      : "/ACKNOWLEDGE",

                example_path_asset    : "/ASSET",
                example_path_text     : "/TEXT",
                example_path_datamap  : "/DATAMAP",

                mainactivity          : "MainActivity",

                a_key                 : "a_value",
                some_other_key        : "some_other_value",
        ]
    }
}
```

Then, to have both modules aware of these key-value pairs and generate the actual code usable in our apps, we need to create a new compilation directive. Hence, for **both** modules (*mobile* and *wear*), edit the respective **build.gradle** accordingly:

```
android {
    ...
    buildTypes {
        ...
        buildTypes.each {
            project.ext.constants.each {
        // - String constants used in Java as `BuildConfig.W_a_key`
        // - Resources are used as usual:
        //    - in Java as:
        //         `[getApplicationContext().]getString(R.string.W_a_key)`
        //    - in XML as:
        //         `@string/W_a_key`
                k, v ->
                    it.buildConfigField 'String', "W_${k}", "\"${v}\""
                    it.resValue 'string', "W_${k}", v
            }
        }
    }
}
```

From now on, in both modules of the project, it will be possible to use a string constant named **W_a_key** which contains the value **W_a_value**, in the way mentioned in comment.

Finally, we need to register the service for the two modules. In the **AndroidManifest.xml**, add the following code in the application element to do three things:

1. Declare the service, as all Android services have to be declared in the manifest file
2. Register the service for Wear Data API events
3. Register the service for Wear Message API events

```
<service android:name=".WearService">
    <intent-filter>
        <action android:name="com.google.android.gms.wearable.DATA_CHANGED" />
        <data
            android:host="*"
            android:pathPrefix=""
            android:scheme="wear" />
    </intent-filter>
    <intent-filter>
        <action android:name="com.google.android.gms.wearable.MESSAGE_RECEIVED" />
        <data
            android:host="*"
            android:pathPrefix=""
            android:scheme="wear" />
    </intent-filter>
</service>
```

5.2 Understanding and using the `WearService` service

Now the service has been added to the project, we will spend some time to understand it, because there are many things in the provided class. Figure 7 will help you follow what is happening from a high level perspective.

Basically, the service uses two facets of the Wear API:

1. The message API, a one-way communication mechanism that's good for remote procedure calls and message passing,
2. The data API, which synchronizes between all connected devices (*nodes*) the data, as if there is a shared memory space. The service we provide you is dealing with two kinds of data:
    1. A **DataMap** is an object which stores key-value associations, just like the **Bundle** objects which can be used for sending data between intents. The main difference is that it will refuse any type which can **not** be sent through the Wear API,
    2. An **Asset** is designed to contain binary data. In the service, we use it to serialize

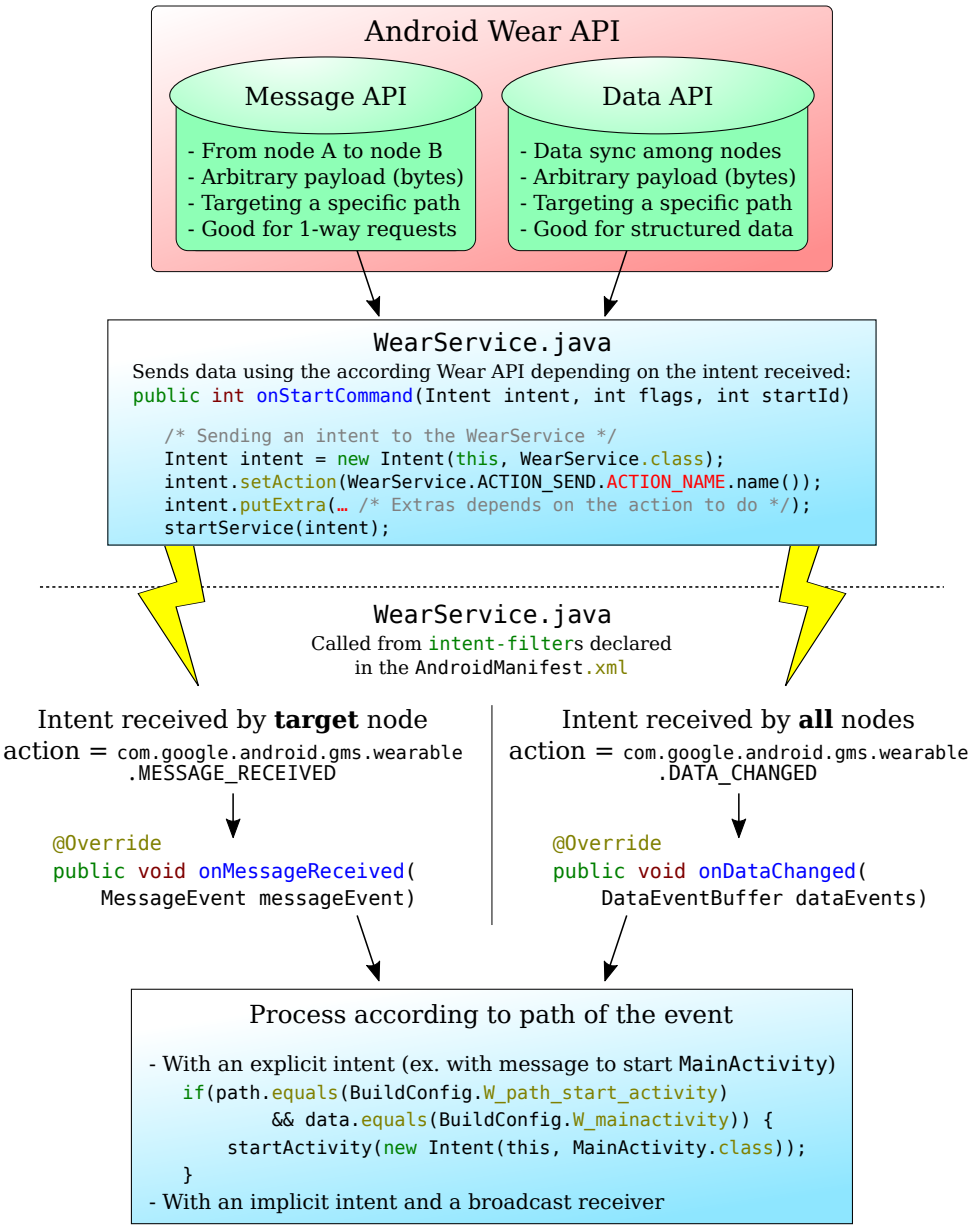

Figure 7: Diagram of the interactions with the **`WearService`**

bitmap (image) data by compressing it as a PNG file, and creating the **`Asset`** from the raw bytes. Reading back the data is the same process in the other way: read and decode the bytes from the **`Asset`** as a PNG file to get the **`Bitmap`** object.

This service is used both for sending, using intents as shown in the first blue box, and receiving, as explained in the lower blue box.

For your own project, you will have to edit both the constants defined in the project's **`build.gradle`**, as well as the **`WearService.java`** file provided. You will have to add and edit

methods to the class to make it fit your needs!

Last but not least, we do not know how to interact with a **Service** yet. To help you with this, we provide four functions to interact with the **WearService** (*it's not mandatory to use it, but highly recommended to test them and see how it works. To send an image you can use the* ***wikipedia_logo.png*** *provided*):

1. **sendStart(...)**: Send message to **W_path_start_activity** path with the content of **W_mainactivity** to start the other device's **MainActivity**
2. **sendMessage(...)**: Send a **DataMap** to **W_example_path_datamap** with:
   1. an integer named **W_a_key** with the value 420
   2. an **ArrayList** named **W_some_other_key** containing the values 105, 107, 109, and 1010
3. **sendDatamap(...)**: Send message to **W_example_path_text** will send back a **DataMap** to **W_example_path_datamap** with:
   1. an integer named **W_a_key** with the value 42
   2. an **ArrayList** named **W_some_other_key** containing the values 5, 7, 9, and 10
4. **sendBitmap(...)**: Send an image as an **Asset** to the **W_example_path_asset** path, with the asset named **W_some_other_key**

Note: Receiving something from the *Data API* such as a **DataMap** or an **Asset** returns back an acknowledgment message which only prints a line in Logcat.

```
public void sendStart(View view) {
    Intent intent = new Intent(this, WearService.class);
    intent.setAction(WearService.ACTION_SEND.STARTACTIVITY.name());
    intent.putExtra(WearService.ACTIVITY_TO_START, BuildConfig.W_mainactivity);
    startService(intent);
}

public void sendMessage(View view) {
    Intent intent = new Intent(this, WearService.class);
    intent.setAction(WearService.ACTION_SEND.MESSAGE.name());
    intent.putExtra(WearService.MESSAGE, "Messaging other device!");
    intent.putExtra(WearService.PATH, BuildConfig.W_example_path_text);
    startService(intent);
}

public void sendDatamap(View view) {
    int some_value = 420;
    ArrayList<Integer> arrayList = new ArrayList<>();
    Collections.addAll(arrayList, 105, 107, 109, 1010);
```

```
    Intent intent = new Intent(this, WearService.class);
    intent.setAction(WearService.ACTION_SEND.EXAMPLE_DATAMAP.name());
    intent.putExtra(WearService.DATAMAP_INT, some_value);
    intent.putExtra(WearService.DATAMAP_INT_ARRAYLIST, arrayList);
    startService(intent);
}

public void sendBitmap(View view) {
    // Get bitmap data (can come from elsewhere) and
    // convert it to a rescaled asset
    Bitmap bmp = BitmapFactory.decodeResource(
            getResources(), R.drawable.wikipedia_logo);
    Asset asset = WearService.createAssetFromBitmap(bmp);

    Intent intent = new Intent(this, WearService.class);
    intent.setAction(WearService.ACTION_SEND.EXAMPLE_ASSET.name());
    intent.putExtra(WearService.IMAGE, asset);
    startService(intent);
}
```

As the service is working in the background, it is using intents to send back the data. We show as an example two intents:

1. An *explicit intent* to start a specific **Activity**, namely the **MainActivity**, in the same way that we already started activities,
2. An *implicit intent* which is broadcast within the application. The **Activity** which needs to receive data from the **WearService** must register to *local broadcasts*, which means that potentially multiple (and unrelated) parts of the application can be updated at once with the newly received data.

Below, a small code snippet showing how to register a listener for image data from a **LocalBroadcastManager**, which you can register in the **onCreate(...)**.

```
LocalBroadcastManager.getInstance(this)
        .registerReceiver(new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.v("MainActivity", "Received intent");
        ImageView imageView = findViewById(R.id.imageView);
        byte[] byteArray = intent.getByteArrayExtra(IMAGE_DATA_BYTES);
        Bitmap bmp = BitmapFactory
```

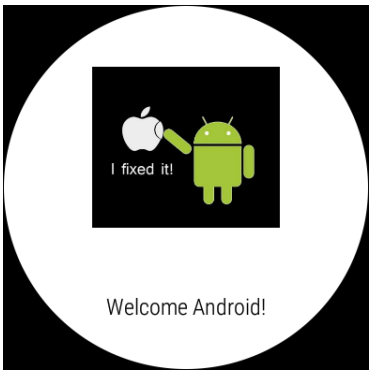

Figure 8: Profile image and username sent to the watch

```
                .decodeByteArray(byteArray, 0, byteArray.length);
        imageView.setImageBitmap(bmp);
    }
}, new IntentFilter(IMAGE_BROADCAST));
```

Is it using the class fields **IMAGE_DATA_BYTES** and **IMAGE_BROADCAST** which you have to declare in the Wear's **MainActivity**, according to the hard-coded strings defined in the **WearService** inviting you to replace them with constants:

```
intent = new Intent("REPLACE_THIS_WITH_A_STRING_OF_ACTION_PREFERABLY_
        DEFINED_AS_A_CONSTANT_IN_TARGET_ACTIVITY");
bitmapFromAsset(asset, intent, "REPLACE_THIS_WITH_A_STRING_OF_IMAGE_
        PREFERABLY_DEFINED_AS_A_CONSTANT_IN_TARGET_ACTIVITY");
```

Now we finished and explained the integration of the **WearService**. Following the example code, add a new **/profile** path, one **photo** and one **username** key-value pair. Now, whenever the **MainActivity** is started and using these newly created constants, first send a message containing the username and display it on the watch. Then, send the image as an **Asset** to replace the logo image on the watch, as shown in Figure 8. Is it possible to send everything in a single step rather than splitting it in two?