Lab 05: Firebase

This lab teaches you how to use Firebase Realtime Database to register the user of our sport tracking app. Additionally, you will use Firebase Storage to save the user image.

1 Introduction to Firebase

Firebase[1] is Google's platform used to provide cloud services for developers, such as storage as well as a real-time database, allowing us to synchronize our app data. It can be used on different platforms offering different products, as shown in Figure 1. Other functionalities are available which can be interesting for your own project (e.g. the user authentication).

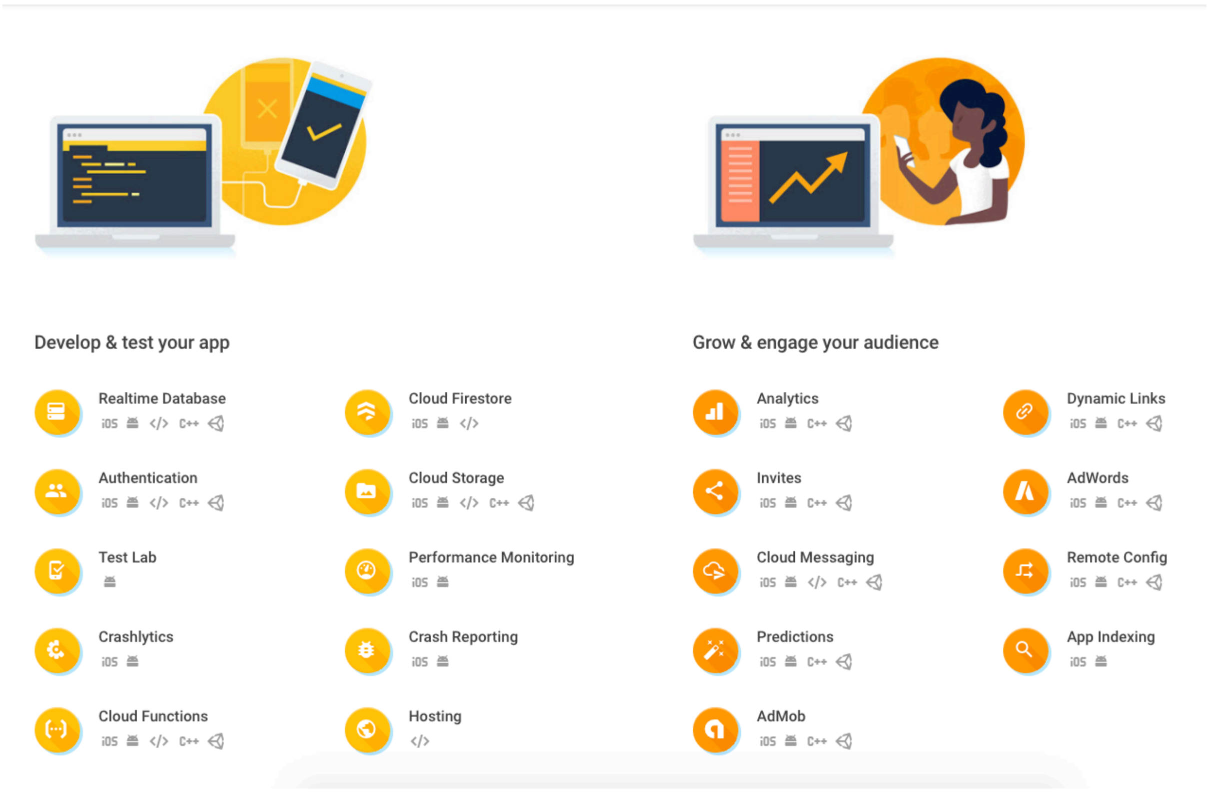


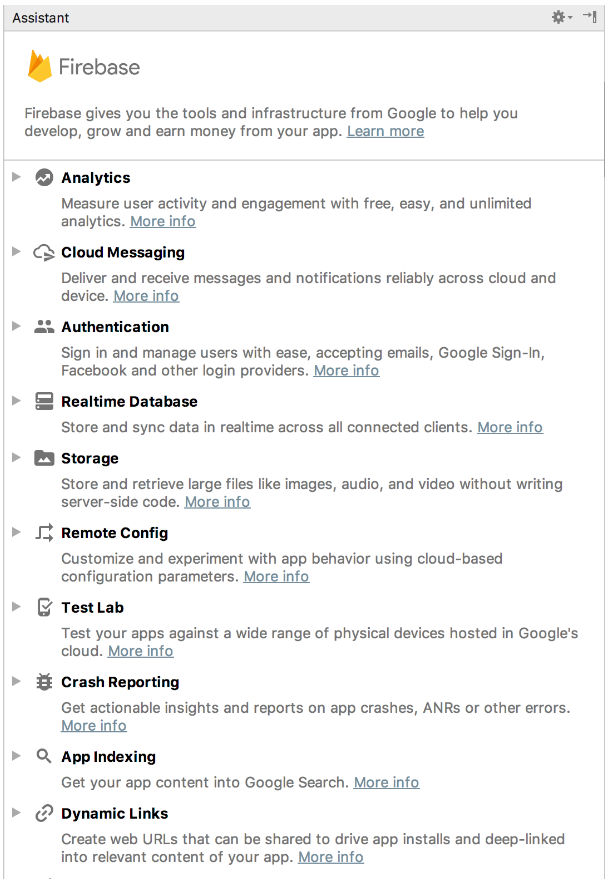
Figure 1: Firebase functionalities

In this lab we will only use the Realtime Database and Storage, to manage different users' profiles. In *Android Studio*, there is a tool called **Firebase Assistant** which helps you connect your app to a Firebase project. First, you need to check if the Firebase Assistant

[1] <https://firebase.google.com/docs/>

is installed by going to **Tools** → **Firebase**. If not, you need to go to **Tools** → **SDK Manager**, switch to the **SDK Tools** tab, and install the **Google Repository**.

Now we have our Firebase Assistant that has all the aforementioned functionalities needed to pair your app. As shown in Fig. 2 and 3, the assistant shows all the steps you have to do:


Assistant

Firebase

Firebase gives you the tools and infrastructure from Google to help you develop, grow and earn money from your app. Learn more

Analytics
Measure user activity and engagement with free, easy, and unlimited analytics. More info

Cloud Messaging
Deliver and receive messages and notifications reliably across cloud and device. More info

Authentication
Sign in and manage users with ease, accepting emails, Google Sign-In, Facebook and other login providers. More info

Realtime Database
Store and sync data in realtime across all connected clients. More info

Storage
Store and retrieve large files like images, audio, and video without writing server-side code. More info

Remote Config
Customize and experiment with app behavior using cloud-based configuration parameters. More info

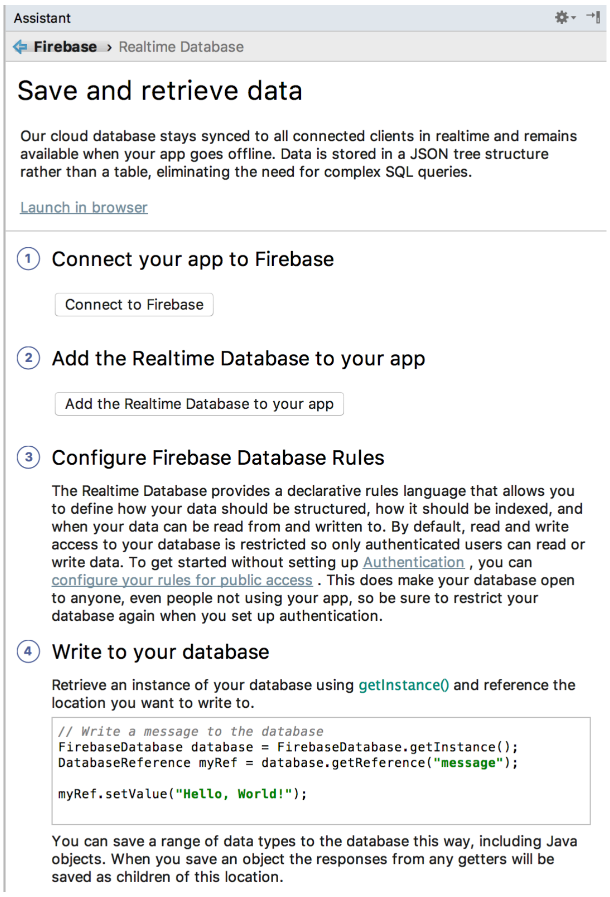
Test Lab
Test your apps against a wide range of physical devices hosted in Google's cloud. More info

Crash Reporting
Get actionable insights and reports on app crashes, ANRs or other errors. More info

App Indexing
Get your app content into Google Search. More info

Dynamic Links
Create web URLs that can be shared to drive app installs and deep-linked into relevant content of your app. More info

Figure 2: Firebase Assistant


Assistant

Firebase › Realtime Database

Save and retrieve data

Our cloud database stays synced to all connected clients in realtime and remains available when your app goes offline. Data is stored in a JSON tree structure rather than a table, eliminating the need for complex SQL queries.

Launch in browser

1 Connect your app to Firebase

Connect to Firebase

2 Add the Realtime Database to your app

Add the Realtime Database to your app

3 Configure Firebase Database Rules

The Realtime Database provides a declarative rules language that allows you to define how your data should be structured, how it should be indexed, and when your data can be read from and written to. By default, read and write access to your database is restricted so only authenticated users can read or write data. To get started without setting up Authentication , you can configure your rules for public access . This does make your database open to anyone, even people not using your app, so be sure to restrict your database again when you set up authentication.

4 Write to your database

Retrieve an instance of your database using getInstance() and reference the location you want to write to.

```
// Write a message to the database
FirebaseDatabase database = FirebaseDatabase.getInstance();
DatabaseReference myRef = database.getReference("message");

myRef.setValue("Hello, World!");
```

You can save a range of data types to the database this way, including Java objects. When you save an object the responses from any getters will be saved as children of this location.

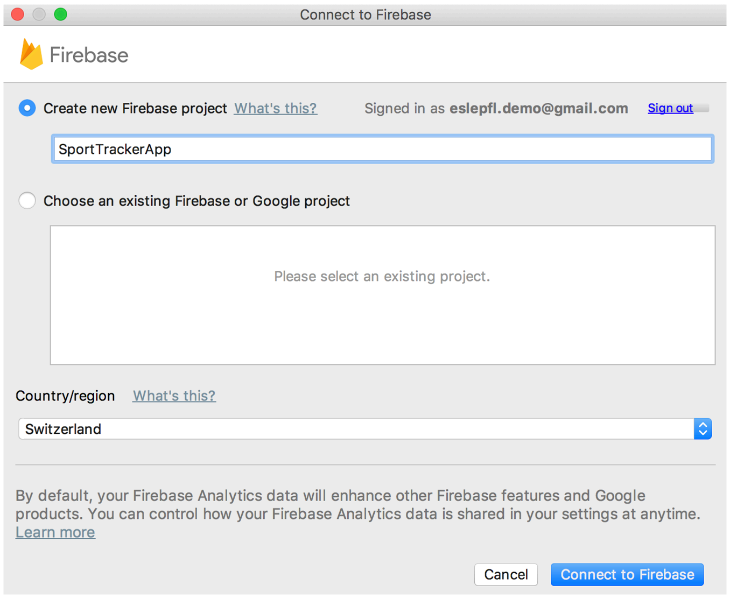
Figure 3: Realtime Database

Let is start by adding the *Realtime Database*:

1. **Connect to Firebase**: you will be requested to create a Firebase Console project that will be attached to your app, as shown in Fig. 4. You only need to choose the module to attach the database, in this case **mobile**. You can access it through the web in Firebase Console with a link that the assistant will provide to you. In case you get an error, just click on the **Connect to Firebase** within the Firebase Assistant and then on **Sync**.
2. **Add the Realtime Database to your app**: in this step the assistant will add some dependencies in the gradle file necessary to attach Firebase Database to your app. Choose **mobile** as targeted module, and then **Accept Changes**. If the build fails, it

is likely that the module's **build**.gradle got mixed up: check that each line in the dependencies block is in the form of compile `com.company.example:library:x.y.z`. At the time of this writing, the dependency line added breaks the compilation and needs to be changed a little: `implementation com.google.firebase:firebase-database:16.0.3`

3. **Configure Firebase Database Rules**: this functionality needs to define data access rules, that is, to specify who can read or write data to and from the database. In this lab, we are not interested in authenticated login, so we configure our rules for public access in the Firebase Console. Note that if you want to use the authentication functionality in your project, Firebase provides it, as well as a simple tutorial in Firebase Assistant. Feel free to explore the Firebase world!



Connect to Firebase

Firebase

(●) Create new Firebase project What's this?   Signed in as eslepfl.demo@gmail.com Sign out

SportTrackerApp

( ) Choose an existing Firebase or Google project

Please select an existing project.

Country/region What's this?

Switzerland

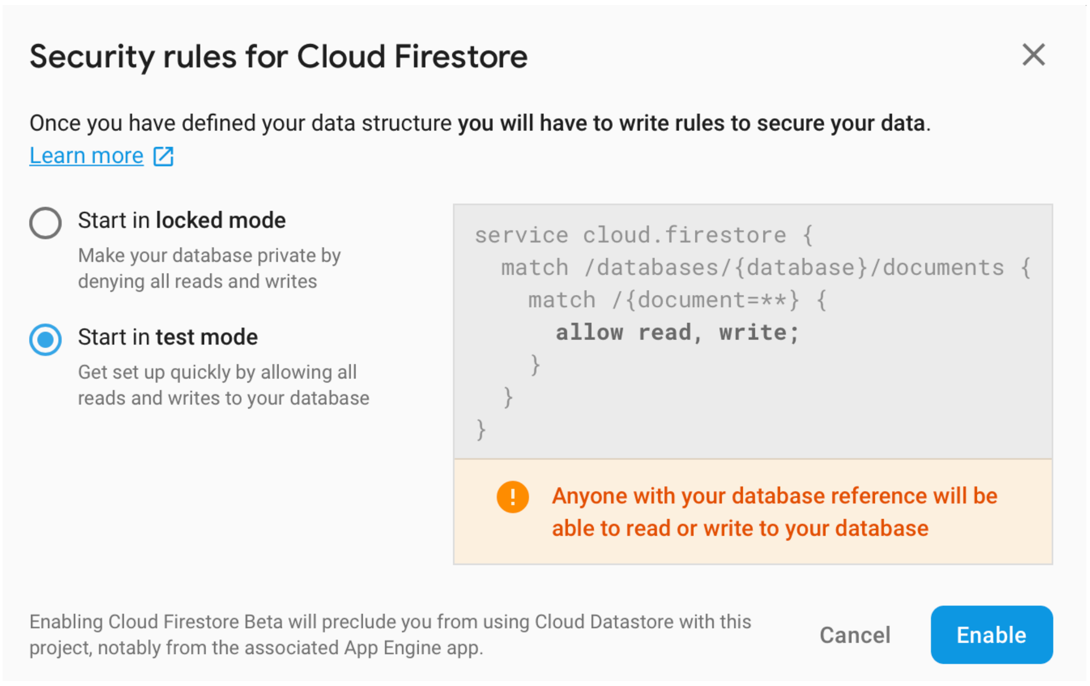
By default, your Firebase Analytics data will enhance other Firebase features and Google products. You can control how your Firebase Analytics data is shared in your settings at anytime. Learn more

Figure 4: Firebase connection

As we want public access, we need to set the rules by going to the Firebase Console[2] at the location of the project you created. Then, enter the project you created, go to **Project Overview** → **Develop** → **Database**, and then **Create Database**. Set the following rules as shown in Fig. 5. Afterwards, select Realtime Database from the dropdown menu in the top bar, switch to the *Rules* tab, and updates the rules to enable *read* and *write* access by changing the values from `false` to `true`.

Furthermore, by following the same process in *Android Studio*, add the *Firebase Storage* from Fig. 2. Once again, be sure the module's **build**.gradle is fine (up-to-date libraries, single specific version). From the *Storage* console, we set the same rules as for the *Realtime Database* (**Project Overview** → **Develop** → **Storage** → **Rules**):

[2]<https://console.firebase.google.com/>


Security rules for Cloud Firestore

Once you have defined your data structure **you will have to write rules to secure your data.**
Learn more

Start in **locked mode** — Make your database private by denying all reads and writes

[✓ Start in **test mode**] — Get set up quickly by allowing all reads and writes to your database

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write;
    }
  }
}
```

Anyone with your database reference will be able to read or write to your database

Enabling Cloud Firestore Beta will preclude you from using Cloud Datastore with this project, notably from the associated App Engine app.

Figure 5: Security rules for Cloud Firestore

```
service firebase.storage {
  match /b/{bucket}/o {
    match /{allPaths=**} {
      allow read, write;
    }
  }
}
```

2 Register user

In Lab 4, in **EditProfileActivity**, we were completing the user profile with a photo along with other data. However, we were not saving the user information anywhere. In this lab, we want to fill the same user form and add the profile to the Firebase Realtime Database, which is based on the JSON syntax. The image profile will be stored in Firebase Storage.

First, we need to add a line of code in the `AndroidManifest.xml` (before the application tag) to give the app permission to access Internet.

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

2.1 Writing to Firebase Realtime Database

We want to save the user profile in the Realtime Database once we click on the **validate** button from the action bar. Therefore, in the **onOptionsItemSelected(...)** function in the **EditProfileActivity**, we add the following function call:

```
addProfileToFirebaseDB();
```

This function writes the user profile into the Realtime Database. We need to implement this method by first getting the instance of the Firebase database. Then, we need to get the reference of the location where we want to put our new profile. In this case, we need to use the **push()** method to create a unique key in the profiles branch that represents a new profile.

```
    private static final FirebaseDatabase database = FirebaseDatabase
            .getInstance();
    private static final DatabaseReference profileGetRef = database
            .getReference("profiles");
    private static final DatabaseReference profileRef = profileGetRef.push();
```

Then, considering the reference to the unique profile key we created, we will run what is called a transaction. Transactions are used in databases to handle writing data all at once rather than change after change. This helps for performance and data consistency. In the overridden method **doTransaction(...)**, there is a **MutableData** object which is the reference to the profile we want to write. We can create children to the profile for each entry of the form: **photo_url**, **username**, **password**, **height**, and **weight**. After writing the values from the profile to the transaction, we need to return the successful transaction and its corresponding **MutableData**. Finally, we need to override the **onComplete(...)** method.

```
private void addProfileToFirebaseDB() {
    profileRef.runTransaction(new Transaction.Handler() {
        @NonNull
        @Override
        public Transaction.Result doTransaction(@NonNull MutableData
                                                        mutableData) {
            mutableData.child("username").setValue(userProfile.username);
            mutableData.child("password").setValue(userProfile.password);
            mutableData.child("height").setValue(userProfile.height_cm);
            mutableData.child("weight").setValue(userProfile.weight_kg);
```

```
                return Transaction.success(mutableData);
            }

            @Override
            public void onComplete(@Nullable DatabaseError databaseError,
                                   boolean b, @Nullable DataSnapshot
                                          dataSnapshot) {

            }
        });
}
```

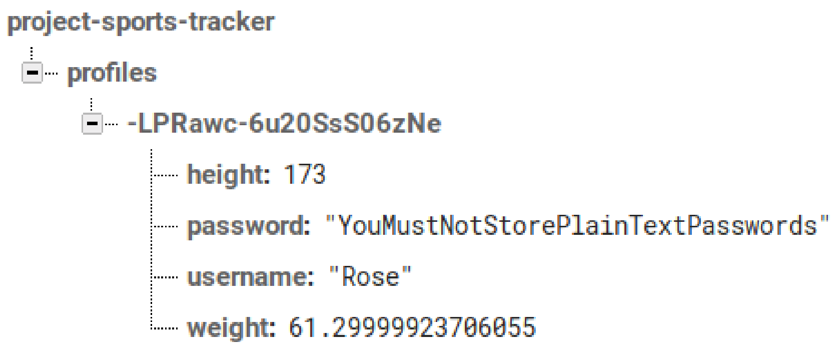
However, to improve the readability of the future code, we will extract the anonymous **`Transaction.Handler() {}`** and give it a proper name. It's easy to do by right-clicking on the **`Handler`** → **Refactor** → **Move...**. You can give this handler the name **`ProfileDataUploadHandler`**, for example. If everything is alright, the method should look like that, now:

```
private void addProfileToFirebaseDB() {
    profileRef.runTransaction(new ProfileDataUploadHandler());
}
```

As we want to save the user profile to the Firebase Realtime Database, the function **`addProfileToFirebaseDB()`** should be invoked from the **`onOptionsItemSelected(...)`**. Therefore, now the function **`onOptionsItemSelected(MenuItem item)`** should look as follows:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_clear:
            clearUser();
            break;
        case R.id.action_validate:
            editUser();
            addProfileToFirebaseDB();
            break;
    }
    return super.onOptionsItemSelected(item);
}
```

The function we implemented in Lab 4 **editUser()** creates a new profile using the user inputs. This profile is then sent to the **addProfileToFirebaseDB()** that writes the profile to the Firebase Realtime Database. You can now run the app and send the user profile to the database. We can see the profile added into the database in Fig. 6.



```
project-sports-tracker
  profiles
    -LPRawc-6u20SsS06zNe
      height: 173
      password: "YouMustNotStorePlainTextPasswords"
      username: "Rose"
      weight: 61.29999923706055
```

Figure 6: Profile added to the Firebase Realtime Database as seen from the Firebase Console. *Note: from a security point of view, never store user passwords in plain-text. Either use the Firebase Auth, either salt and hash properly the password before sending it online.*

2.2 Toasts

A toast provides simple feedback about an operation in a small popup, and it automatically disappears after a timeout. In order to illustrate the use of toasts, let us have a look at the function **onComplete(...)**. Namely, this method will be called once with the results of the transaction. The input parameters of this method are the following ones:

1. **DatabaseError databaseError**: **null** if no errors occurred, otherwise it contains a description of the error.
2. **boolean b**: **true** if the transaction successfully completed, **false** otherwise
3. **DataSnapshot dataSnapshot**: the current data in the database, possibly **null**

In case the transaction is successfully completed we get a toast notification of success, otherwise we get a message that the registration was not successfully completed.

```
if (b) {
    Toast.makeText(EditProfileActivity.this, R.string
            .registration_success, Toast.LENGTH_SHORT).show();
} else {
    Toast.makeText(EditProfileActivity.this, R.string
            .registration_failed, Toast.LENGTH_SHORT).show();
}
```

The **Toast.makeText(...)** method takes three parameters:

1. the application **Context** which is in our case **EditProfileActivity.this**,
2. the text message, as a **String** or directly a *string ressource*
3. the duration for the toast.

It returns a properly initialized **Toast** object, which finally only requires to be displayed to the user by calling **show()**.

2.3 Uploading user image to Firebase Storage

The profile is still not complete as we miss the profile image. For this purpose we need to use the Firebase Storage. We will add the uploading part in the same **addProfileToFir⌋ ebaseDB()** method used to write data to the Firebase Realtime Database.

However, we will **first** upload the image to the Storage, retrieve its downloadable link, and **then** add this information to the Realtime Database.

The first thing to do is to get the image from the **View** (if not missing) and convert it into raw bytes to be able to upload it:

```
BitmapDrawable bitmapDrawable = (BitmapDrawable) ((ImageView) findViewById(R.id
        .userImage)).getDrawable();
if (bitmapDrawable == null) {
    Toast.makeText(this, R.string.missing_picture, Toast
            .LENGTH_SHORT).show();
    return;
}
Bitmap bitmap = bitmapDrawable.getBitmap();
ByteArrayOutputStream baos = new ByteArrayOutputStream();
bitmap.compress(Bitmap.CompressFormat.JPEG, 90, baos);
byte[] data = baos.toByteArray();
```

Now we have the raw bytes, we can upload the image, giving it the name of the profile's *key* from the Realtime Database:

```
StorageReference storageRef = FirebaseStorage.getInstance()
        .getReference();
StorageReference photoRef = storageRef.child("photos").child
        (profileRef.getKey() + ".jpg");
UploadTask uploadTask = photoRef.putBytes(data);
uploadTask.addOnFailureListener(new OnFailureListener() {
    @Override
    public void onFailure(@NonNull Exception exception) {
        // Handle unsuccessful uploads
        Toast.makeText(EditProfileActivity.this, R.string
                .photo_upload_failed, Toast.LENGTH_SHORT).show();
    }
}).addOnSuccessListener(new OnSuccessListener<UploadTask
        .TaskSnapshot>() {
    @Override
    public void onSuccess(UploadTask.TaskSnapshot taskSnapshot) {

    }
});
```

As we will add quite a bit of code in the **OnSuccessListener<>**, you can extract it in the same way we already did for the **Transaction.Handler** previously, with a right-click on **OnSuccessListener** → Refactor → Move... and giving it a meaningful name such as **PhotoUploadSuccessListener**. It will suggest to make it **static** but it will not fit our needs as it will require access to non-static methods.

This newly created listener is called whenever the image upload finishes successfully. It means it is now time to get its downloadable URL, and set it in the **userProfile.photoPath** field. It is once again an asynchronous process which uses listeners:

```
private class PhotoUploadSuccessListener implements
        OnSuccessListener<UploadTask.TaskSnapshot> {
    @Override
    public void onSuccess(UploadTask.TaskSnapshot taskSnapshot) {
        Task<Uri> downloadUrl = taskSnapshot.getMetadata().getReference()
                .getDownloadUrl().addOnSuccessListener(new OnSuccessListener<Uri>() {
            @Override
            public void onSuccess(final Uri uri) {
                userProfile.photoPath = uri.toString();
            }
        });
    }
}
```

At this point, when the user saves the profile, we get the image from the **View**, compress it to bytes as JPEG, upload it, retrieve its address for downloading it in the future. We finally need to save the user's data. We were doing it initially in the **addProfileToFirebase()** method, but we will move it to the **onSuccess()** listener **Task** named **downloadUrl**:

```
@Override
public void onSuccess(final Uri uri) {
    userProfile.photoPath = uri.toString();
    profileRef.runTransaction(new ProfileDataUploadHandler());
}
```

Do not forget to edit the **ProfileDataUploadHandler** to *also* save the photo URI by adding one new child to the **mutableData**:

```
mutableData.child("photo").setValue(userProfile.photoPath);
```

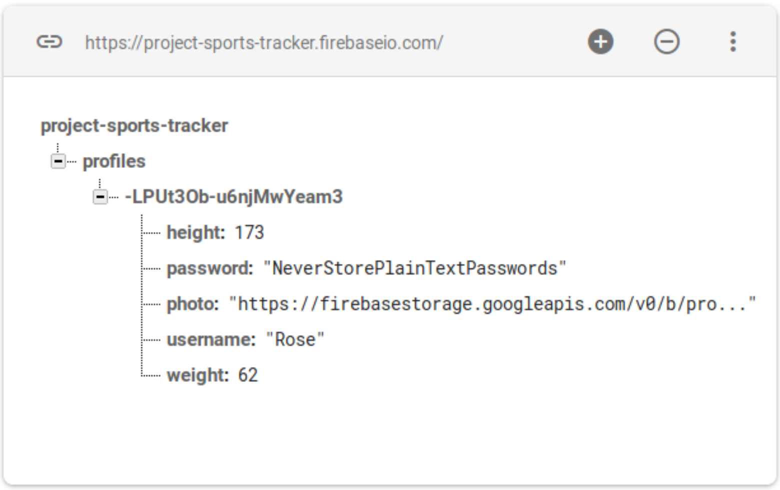
Finally, to be consistent with the new workflow with Firebase, we need to change a little the **editUser()** method. As a matter of fact, we should not leave the **EditProfileActivity** if the full uploading process has not been finished. We can therefore the following piece of code in the **onComplete()** callback of our **ProfileDataUploadHandler**:

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

If you try the app now, you should get the Firebase Console for the Realtime Database similar to Fig. 7 and the Storage one similar to Fig. 8.

3 Login user

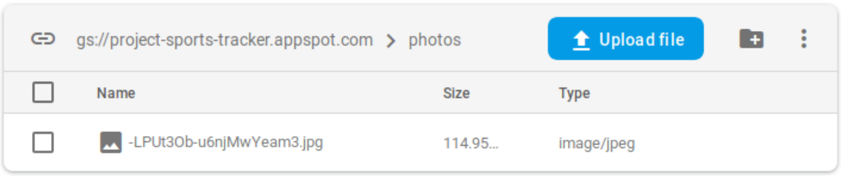
In this part, we need to check if the user is already a member of our database. In order to do this, we need to read the data from the Realtime Firebase Database. To read the data, you need to attach the **ValueEventListener()** to the database reference. This event is triggered whenever there is a change in data in realtime. Implementation of the **ValueEventListener()** requires the following two functions to be overriden: **onDataChange(...)** and **onCancelled()**. The function **onDataChange(...)** is called each time that data changes, whereas the function **onCancelled(...)** is triggered in the event that this listener either failed at the server, or is removed as a result of the security and Firebase Database rules.



https://project-sports-tracker.firebaseio.com/

```
project-sports-tracker
  profiles
    -LPUt3Ob-u6njMwYeam3
      height: 173
      password: "NeverStorePlainTextPasswords"
      photo: "https://firebasestorage.googleapis.com/v0/b/pro..."
      username: "Rose"
      weight: 62
```

Figure 7: Realtime Database entry with the link to the downloadable profile image.



gs://project-sports-tracker.appspot.com > photos

Name	Size	Type
-LPUt3Ob-u6njMwYeam3.jpg	114.95...	image/jpeg

Figure 8: Storage Database entry with profile image.

We read the data from the Realtime Firebase Database (profile by profile), and we compare user's input (username and password) to those of registered users. In case of registered users, we send the unique user's key, obtained through **getKey()** to the **MainActivity** class through an intent that will further be used for reading the entire user information from the Realtime Firebase Database in **MyProfileFragment**. The **onClick** function related to the *Login* button press, should look as follows:

```
final FirebaseDatabase database = FirebaseDatabase.getInstance();
final DatabaseReference profileRef = database.getReference("profiles");

final TextView mTextView = findViewById(R.id.LoginMessage);
final String usernameInput = ((EditText) findViewById(R.id.username)).getText().toStrin
final String passwordInput = ((EditText) findViewById(R.id.password)).getText().toStrin

profileRef.addValueEventListener(new ValueEventListener() {
    @Override
    public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
        boolean notMember = true;
        for (final DataSnapshot user : dataSnapshot.getChildren()) {
            String usernameDatabase = user.child("username").getValue(String.class);
            String passwordDatabase = user.child("password").getValue(String.class);
            if (usernameInput.equals(usernameDatabase) && passwordInput.equals
                    (passwordDatabase)) {
                userID = user.getKey();
                notMember = false;
                break;
            }
        }
        if (notMember) {
            mTextView.setText(R.string.not_registered_yet);
            mTextView.setTextColor(Color.RED);
        } else {
            Intent intent = new Intent(LoginActivity.this, MainActivity.class);
            intent.putExtra(MyProfileFragment.USER_ID, userID);
            startActivity(intent);
        }
    }

    @Override
    public void onCancelled(@NonNull DatabaseError databaseError) {
    }
});
```

Note: We are here doing a trivial implementation of a login feature for learning how to use the Firebase Realtime Database. However, it has a lot of shortcomings, the two most important are:

1. All the passwords are stored in plain-text
2. All the database is downloaded for each user, which is leaking all passwords as well as not being able to scale well above few hundreds of users.

Once again, if you need a real authentication, please use the Firebase Auth API.

3.1 Reading from Firebase Realtime and Storage databases

In case of a registered user, the unique key is sent to the **MainActivity** class through an **Intent**. We extract this key from the intent in **MyProfileFragment** and we use it to read the entire user profile from the Firebase RealTime Database. We extract the intent data in the **onCreateView(...)** within **MyProfileFragment** as follows:

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
    // Inflate the layout for this fragment
    fragmentView = inflater.inflate(R.layout.fragment_my_profile,
            container, false);

    Intent intent = getActivity().getIntent();
    userID = intent.getExtras().getString(USER_ID);
    readUserProfile();

    return fragmentView;
}
```

The **`readUserProfile()`** is a method that reads the user's profile and writes it in the **`MyProfileFragment`**. The same way we did it for login, we first get a reference to the database. From here we need to use the child node names to traverse further. We use function to read the values of different fields. As previously explained, we attach a **`addValueEventListener(...)`** and override functions **`onDataChange(...)`** and **`onCancelled(...)`**.

```
private void readUserProfile() {
    final FirebaseDatabase database = FirebaseDatabase.getInstance();
    final DatabaseReference profileRef = database.getReference("profiles");
    profileRef.child(userID).addValueEventListener(new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
            String user_db = dataSnapshot.child("username").getValue(String.class);
            String password_db = dataSnapshot.child("password").getValue(String.class);
            int height_db = dataSnapshot.child("height").getValue(int.class);
            float weight_db = dataSnapshot.child("weight").getValue(float.class);
            String photo = dataSnapshot.child("photo").getValue(String.class);

            userProfile = new Profile(user_db, password_db);
            userProfile.password = password_db;
            userProfile.height_cm = height_db;
            userProfile.weight_kg = weight_db;
            userProfile.photoPath = photo;

            setUserImageAndProfileInfo();
        }
        @Override
        public void onCancelled(@NonNull DatabaseError databaseError) {
            // Empty
        }
    });
}
```

We now need to update the **setUserImageAndProfileInfo()** according to reading the image info from Firebase. It is overall very similar as previously, except we set all the image data in the **onSuccessListener** of the Firebase Storage call:

```
private void setUserImageAndProfileInfo() {
    //  Reference to an image file in Firebase Storage
    StorageReference storageRef = FirebaseStorage.getInstance()
            .getReferenceFromUrl(userProfile.photoPath);
    storageRef.getBytes(Long.MAX_VALUE).addOnSuccessListener(
            new OnSuccessListener<byte[]>() {
        @Override
        public void onSuccess(byte[] bytes) {
            if (isAdded()) {
                final Bitmap selectedImage = BitmapFactory
                        .decodeByteArray(bytes, 0, bytes.length);
                ImageView imageView = fragmentView.findViewById(R.id
                        .userImage);
                imageView.setImageBitmap(selectedImage);
            }
        }
    });

    TextView usernameTextView = fragmentView.findViewById(R.id.usernameValue);
    usernameTextView.setText(userProfile.username);

    TextView passwordTextView = fragmentView.findViewById(R.id.passwordValue);
    passwordTextView.setText(userProfile.password);

    TextView heightTextView = fragmentView.findViewById(R.id.heightValue);
    heightTextView.setText(String.valueOf(userProfile.height_cm));

    TextView weightTextView = fragmentView.findViewById(R.id.weightValue);
    weightTextView.setText(String.valueOf(userProfile.weight_kg));
}
```

3.2 Updating user profile

The user can update the profile information by clicking on the editing button in the **ActionBar** menu. Once the button is clicked, the we display the current user profile (from the Firebase Realtime Database). In this case, we have to send the user's unique key to the **EditProfileActivity** through an intent. This is done in the function **onOptionsItemSelected(...)** in the **MyProfileFragment**:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_edit:
            Intent intentEditProfile = new Intent(getActivity(),
                    EditProfileActivity.class);
            intentEditProfile.putExtra(USER_ID, userID);
            startActivityForResult(intentEditProfile, EDIT_PROFILE_INFO);
            break;
    }

    return super.onOptionsItemSelected(item);
}
```

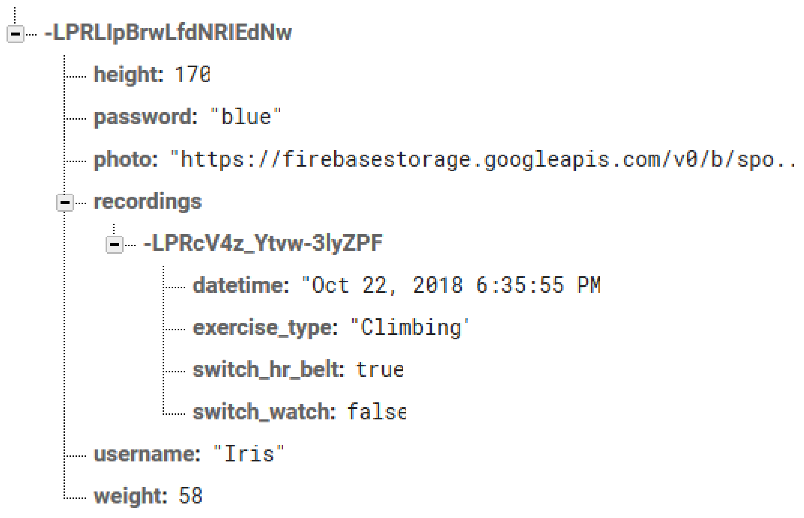
In the **onCreate(...)** function of the **EditProfileActivity**, we need to extract the intent data we sent from **MyProfileFragment** (user's unique key).

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_edit_profile);

    Intent intent = getIntent();
    if (intent.getExtras() != null) {
        userID = intent.getExtras().getString(USER_ID);
        fetchDataFromFirebase();
    }

}
```

The function **fetchDataFromFirebase()** reads the profile of the user whose unique key we sent from the **MyProfileFragment**. If we upload data over the same key, the content is replaced accordingly, so we only need to set the **profileRef** to the right Firebase entry:

```java
private void fetchDataFromFirebase() {
    final TextView usernameTextView = findViewById(R.id.editUsername);
    final TextView passwordTextView = findViewById(R.id.editPassword);
    final TextView heightTextView = findViewById(R.id.editHeight);
    final TextView weightTextView = findViewById(R.id.editWeight);

    profileGetRef.child(userID).addValueEventListener(new ValueEventListener() {
        @Override
        public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
            String user_db = dataSnapshot.child("username").getValue(String.class);
            String password_db = dataSnapshot.child("password").getValue(String.class);
            int height_db = dataSnapshot.child("height").getValue(int.class);
            float weight_db = dataSnapshot.child("weight").getValue(float.class);

            usernameTextView.setText(user_db);
            passwordTextView.setText(password_db);
            heightTextView.setText(String.valueOf(height_db));
            weightTextView.setText(String.valueOf(weight_db));

            profileRef = profileGetRef.child(userID);
        }

        @Override
        public void onCancelled(@NonNull DatabaseError databaseError) {

        }
    });
}
```

In the same way we read the image from Firebase in the **MyProfileFragment** in the **setUserImageAndProfileInfo()** call, you can retrieve the bitmap and set it in the **ImageView**



```
-LPRLIpBrwLfdNRIEdNw
    height: 170
    password: "blue"
    photo: "https://firebasestorage.googleapis.com/v0/b/spo..
    recordings
        -LPRcV4z_Ytvw-3lyZPF
            datetime: "Oct 22, 2018 6:35:55 PM
            exercise_type: "Climbing'
            switch_hr_belt: true
            switch_watch: false
    username: "Iris"
    weight: 58
```

Figure 9: JSON tree for a specific user with a new recording child

4 Record exercises and show history

On top of the user profile information, we want to save our new recordings that we will make with our sport tracker. Additionally, we want to show the history of our recordings. For now, we will only save and show the description of the exercise we are starting, while in the next labs we will be able to access sensors and record a real exercise.

4.1 Save recording info in Firebase

The steps to write our new recording exercise in Firebase is the same as seen in Section 2.1. We will give you some hints/steps on how to proceed but you have all the tools to implement this part at this point! These are the steps you should follow:

1. Insert a button in `fragment_new_recording.xml` with the id `saveNewRecButton`. This will be our button to save the new recording on Firebase
2. In the `onCreateView(...)` of `NewRecordingFragment` set the `onClickListener` for this button and call a new private method where you can implement (later) the loading of the new recording on Firebase. You can call the method as you prefer or you can also implement everything within the `onClickListener`
3. Since we want to link the recording to the specific user logged in, we need to get the user's unique key sent to the `MainActivity` through the `Intent`. Implement it as we have already shown in previous sections.
4. Now, that we have the Firebase user's unique key we can add the recording as shown in Figure 9. What is our object `DatabaseReference` in this case?

4.2 Read past recordings from Firebase

We can read our history of recordings from Firebase, but first we want to prepare the layout of **MyHistoryFragment**. We will replace the layout of **fragment_my_history.xml** and place two **View**s in a vertical **LinearLayout**: a **ListView** with the id *myHistoryList* and a **FrameLayout** with id *plotRecording*. Put the weight of the two views at 1, so that we will have half of the screen reserved to each view. The first view belongs to the view group **ListView**, which displays a list of scrollable items. In our case, we will have a list of recordings with their description that we will read from Firebase. The second view is just a space we will reserve for a future plot of the data acquired from our exercises (wait for next labs!).

Now, let's focus on the **ListView** which will contain our recordings. For this purpose, we need to create one layout for the item of the list, since each item will have the same elements. Create the layout in the **res** folder called to **row_myhistory_layout.xml**. Use the layout you prefer with four **TextView**s with the following ids: *exerciseType*, *exerciseDateTime*, *exerciseDevice*, *exerciseDevice2*. Additionally, we can create a class **Recording** similar to the **Profile** one we already have with the **protected** fields: a **String exerciseType**, a **long exerciseDateTime** (time in millisecond), a **boolean exerciseSmartWatch** and a **boolean exerciseHRbelt**.

Our **ListView** will contain a list of **Recording** objects and for this we need an **ArrayAdapter**, which returns a view for each object in the collection of data objects we provide, in our case the recordings. Since we created the **Recording** class, the **ArrayAdapter** doesn't know how to handle these type of objects, so we need to create our own adapter. We will create an inner class (a private class inside **MyHistoryFragment**) implemented as shown in the code below.

```java
private class RecordingAdapter extends ArrayAdapter<Recording> {

    private int row_layout;

    RecordingAdapter(FragmentActivity activity, int row_layout) {
        super(activity, row_layout);
        this.row_layout = row_layout;
    }

    @NonNull
    @Override
    public View getView(int position, @Nullable View convertView,
                        @NonNull ViewGroup parent) {
```

```
        //Reference to the row View
        View row = convertView;
        if (row == null) {
            //Inflate it from layout
            row = LayoutInflater.from(getContext()).inflate(row_layout,
                    parent, false);
        }

        SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy " +
                "hh:mm:ss", Locale.getDefault());

        ((TextView) row.findViewById(R.id.exerciseType)).setText(getItem
                (position).exerciseType);
        ((TextView) row.findViewById(R.id.exerciseDateTime)).setText
                (formatter.format(new Date(getItem(position)
                        .exerciseDateTime)));
        ((TextView) row.findViewById(R.id.exerciseDevice)).setText
                (getString(R.string.smartwatch_switch_value, getItem
                        (position).exerciseSmartWatch ? "yes" : "no"));
        ((TextView) row.findViewById(R.id.exerciseDevice2)).setText
                (getString(R.string.hr_belt_switch_value, getItem
                        (position).exerciseHRbelt ? "yes" : "no"));
        return row;
    }
}
```

The class **RecordingAdapter** constructor takes in input the **Activity** where the fragment is attached and the id of the row layout. Then, we need to override the method **getView(...)** as this is the one called when adding items to the visible list. In this method we need to set the view of the row layout with the data we want. In our case, the data is coming from the recording objects, in particular from its fields.

After this, we need to get the **ListView** from the fragment layout and set the newly created adapter for this view. We do this in the **onCreateView(...)** of **MyHistoryFragment** by adding the following code. Remember to add the fields to the class if needed.

```
listView = fragmentView.findViewById(R.id.myHistoryList);
adapter = new RecordingAdapter(getActivity(), R.layout
        .row_myhistory_layout);
listView.setAdapter(adapter);
```

At this point, we know how to handle the list of recordings but we are not yet filling the list with actual data. It's time to get our recordings from **Firebase**!

From time to time, we were getting data from Firebase by using an *anonymous* implementation of **`ValueEventListener`**, which does not have a name and lets you to declare and instantiate a class at the same time and use it only once. This is how it looked:

```
someRef.addValueEventListener(new ValueEventListener() {
    // Some methods to override
    // Some more code as a normal class would be implemented
});
```

In this case, particularly in **`MyHistoryFragment`**, we want to add (*register*) and remove (*unregister*) the listener based on the lifecycle of the fragment, that is in the **`onResume()`** and **`onPause()`** methods. In order to do this, we need to create our own **`MyFirebaseRecordingListener`** to specifically listen for events related to our recordings added in Firebase. We add this listener as an inner class as we did for the **`ArrayAdapter`**.

```
private class MyFirebaseRecordingListener implements ValueEventListener {
    @Override
    public void onDataChange(DataSnapshot dataSnapshot) {
        adapter.clear();

        for (final DataSnapshot rec : dataSnapshot.getChildren()) {
            final Recording recording = new Recording();

            recording.exerciseType = rec.child("exercise_type")
                    .getValue().toString();
            recording.exerciseDateTime = Long.parseLong(rec.child
                    ("datetime").getValue().toString());
            recording.exerciseSmartWatch = Boolean.parseBoolean(rec
                    .child("switch_watch").getValue().toString());
            recording.exerciseHRbelt = Boolean.parseBoolean(rec
                    .child("switch_hr_belt").getValue().toString());

            adapter.add(recording);
        }
    }

    @Override
    public void onCancelled(DatabaseError databaseError) {
        Log.v(TAG, databaseError.toString());
    }
}
```

The listener overrides the method **onDataChange(...)** as it was done with the anonymous class. Here we get the data from Firebase at the tree level of **recordings** if we refer to Figure 9. Moreover, we need to add the **Recording** object we retrieved into the adapter. Notice that before looking into all recordings we always clear the adapter otherwise it will add the data every time we listens for events.

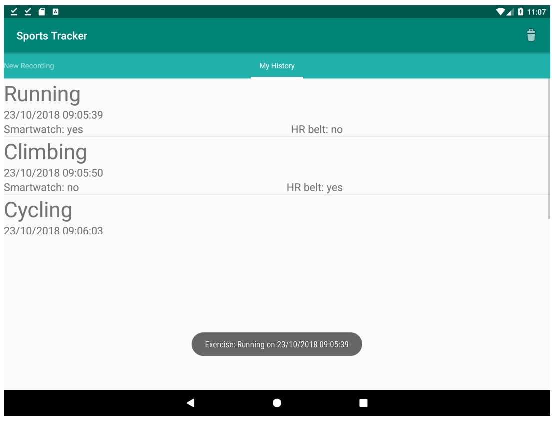
Next, we need to override the methods **onResume()** and **onPause()** to register and unregister the listener. We do this because reading from an external source such as Firebase, might create problems if active fragments are trying to change UI elements while being detached because of a configuration change (tablet rotation for example, see Section 5). The code below shows what to add.

```
@Override
public void onResume() {
    super.onResume();
    databaseRef = FirebaseDatabase.getInstance().getReference();
    mFirebaseRecordingListener = new MyFirebaseRecordingListener();
    databaseRef.child("profiles").child(idUser).child("recordings")
            .addValueEventListener(mFirebaseRecordingListener);
}
```

In the **onResume()**, we get the reference to the Firebase real time database considering the specific user logged in (you can get the user unique key through an **Intent** as done in the other fragments) and then we create a new **MyFirebaseRecordingListener**. Finally, we add this listener to the database reference.

```
@Override
public void onPause() {
    super.onPause();
    databaseRef.child("profiles").child(idUser).child("recordings")
            .removeEventListener(mFirebaseRecordingListener);
}
```

In the **onPause()**, we unregister the listener to avoid memory leaks: if we keep the listener registered, the existing list are kept in memory on orientation change while only the newly created list is used and displayed. This is particularly visible if you load images in the list retrieved from **Firebase** since images takes a big space in memory.


Sports Tracker

New Recording My History

Running
23/10/2018 09:05:39
Smartwatch: yes HR belt: no

Climbing
23/10/2018 09:05:50
Smartwatch: no HR belt: yes

Cycling
23/10/2018 09:06:03

Exercise: Running on 23/10/2018 09:05:39

Figure 10: List of recordings in MyHistoryFragment

Try to add the image of the exercise type that we show in **NewRecordingFragment** in the row layout of **MyHistoryFragment** list view for a nicer looking history!

Finally, we want to do something when clicking on item of the list and for this we need to call the method **setOnItemClickListener(...)** on the list view. We do this in the **onCreateView(...)** of **MyHistoryFragment** after setting the adapter.

```
listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> adapterView, View view,
                            int i, long l) {
        Toast.makeText(getContext(), ”Exercise: ” + ((TextView) view
                .findViewById(R.id.exerciseType)).getText().toString
                () + ” on ” + ((TextView) view.findViewById(R.id
                .exerciseDateTime)).getText().toString(), Toast
                .LENGTH_SHORT).show();
    }
});
```

The list view with the toast shown at the item clicked should look like the one in Figure 10.

Try to implement the **Delete history** menu item now that you know how to use Firebase (*hint:* you can use the method **removeValue()**).

5 Handling configuration changes

One last thing we want to check is what happens to the data of our user profile when we turn the orientation of our tablet/smartphone. If you try to do that in **EditProfileActivity** for example, you will notice that the user image disappears. This happens because the **Activity** is destroyed and created again in what is called *configuration change*. One method to handle configuration changes is to save the resources that can be lost, such as our user image, in a **Bundle** object used for this specific purpose. We do this by overriding a method called **onSaveInstanceState(...)** where we will save the **Uri** of the image we get from the **Intent** with **ACTION_GET_CONTENT** in the **Bundle** object. We can create a private field to the class called **savedImageUri** to save when we get the content from the **Intent** mentioned before.

```
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    outState.putParcelable("ImageUri", savedImageUri);
}
```

Since the **Activity** is destroyed and created again we need to retrieve our user image within the method **onCreate(...)** (or **onCreateView(...)** if we are working with fragments).

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    if (savedInstanceState != null) {
        savedImageUri = savedInstanceState.getParcelable("ImageUri");
        if (savedImageUri != null) {
            // Put the image in the ImageView,
            // as you did in onActivityResult(…)
        }
    }
}
```

Now, run the app and try to change the orientation of your tablet/smartphone and the image will not disappear anymore!

If we are editing the profile calling **EditProfileActivity** from **MyProfileFragment**, what do we change/add to keep the user image during a configuration change?