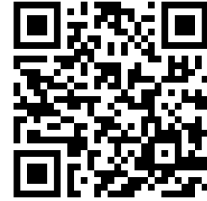


Assignment 6

Welcome to another week of Mobile Systems and Applications! You can also access this assignment in the most up-to-date format @ cs.upt.ro/~alext/msa/lab6



Topics of discussion

- Data storage **options**
- Introduction to Google **Firebase**: real-time database with NoSQL

Contents

1. Data storage options	1
1.1. Saving to Shared preferences	1
1.2. Saving to files	3
1.3. Saving to a database	4
2. Google Firebase	5
2.1. Project setup	5
3. Task #1	11
4. Task #2	16

1. Data storage options

Most Android apps need to save data, even if only to save information about the app state during *onPause* so the user's progress is not lost. Most non-trivial apps also need to save user settings, and some apps must manage large amounts of information in files and databases. As such, we will talk about the main data storage options available in Android:

- Shared preferences: saving key-value sets
- Files on internal (device memory or shared external storage)
- Databases: SQLite and other network connections

1.1. Saving to Shared preferences

If you have a relatively small collection of key-values that you'd like to save, you should use *SharedPreferences*. A *SharedPreferences* object points to a file containing key-value pairs and provides simple methods to read and write them. Each *SharedPreferences* file is managed by the application, and can be private or shared. A private file can be created using the following simple code:

```

private final static String PREFS_SETTINGS = "prefs_settings";
private SharedPreferences prefsUser, prefsApp;

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

    // named preference file
    prefsUser = getSharedPreferences(PREFS_SETTINGS, Context.MODE_PRIVATE);
    // default prefs file for this app
    prefsApp = getPreferences(Context.MODE_PRIVATE);
}

```

In the code above we declare two preference files as class members of the activity and instantiate them in *onCreate*. The first instantiation (*prefsUser*) creates a new named shared preferences file that is private to the application, i.e. it cannot be seen by other applications. The second instantiation (*prefsApp*) creates an unnamed default preference file that is accessible to this app. On a normal basis, it is advised to use named files so that you may structure and delimit different types of data you may want to save. You may also create public preferences using the options `MODE_WORLD_READABLE` or `MODE_WORLD_WRITEABLE`; these have become deprecated nonetheless.

In order to write and read data from a shared preference we need the following methodology: writing is done by acquiring an editor on the preference file, then writing the key-value pair, followed by a commit. The types of data that may be written are: integer, float, long, boolean, string and a string set. To write each value you need to define a key under which it is stored. This key is usually defined as a final static String and must be unique, otherwise the values will be overwritten. In the example below we acquire an editor, write four different values under keys KEY1-KEY4, and then commit the changes. In the second example, a long value is committed using one single line of code.

```
// named preference file
prefsUser = getSharedPreferences(PREFS_SETTINGS, Context.MODE_PRIVATE);

SharedPreferences.Editor editor = prefsUser.edit();
editor.putInt("KEY1", 10);
editor.putString("KEY2", "hello there");
editor.putBoolean("KEY3", true);
editor.putFloat("KEY4", 3.1415f);
editor.commit();

// one-line code for writing data to file
prefsUser.edit().putLong("KEY5", 100022L).apply();
```

Remember that only once you commit the changes will the data be written in the file! Committing changes can be done in two ways: synchronous (using *editor.commit*) or asynchronous (using *editor.apply*). The second method call is encouraged.

Reading is done in a similar way. While we don't need an editor to read the data, we need to specify a default value to be returned in case no value is found under the specified key. For example, the code below demonstrates how we try to read an integer and save it's value in the variable *score*. If "KEY1" does not exist, then the default value 0 is returned. Similar, if "KEY2" is not found, then the string *name* will have the default value *null*.

```
// get value (or default value) from prefs
int score = prefsUser.getInt("KEY1", 0);
String name = prefsUser.getString("KEY2", null);
```

1.2. Saving to files

Android uses a file system that's similar to disk-based file systems on other platforms. All Android devices have two file storage areas: "internal" and "external" storage. Some devices divide the permanent storage space into "internal" and "external" partitions, so even without a removable storage medium, there are always two storage spaces and the API behavior is the same whether the external storage is removable or not.

Facts about internal storage: it's always available; files saved here are accessible by only your app; when the user uninstalls your app, the system removes all your app's files from internal storage.

Facts about external storage: it's not always available, because the user can mount the external storage as USB storage and in some cases remove it from the device; it's world-readable, so files saved here may be read outside of your control; when the user uninstalls your app, the system removes your app's files from here only if you save them in the directory from *getExternalFilesDir*.

In order to read and write files from internal storage we must first obtain the permission to do so. This is done by requesting the appropriate permissions in the manifest file.

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

When saving a file to internal storage, you need to acquire the appropriate directory as a File by calling one of two methods: *getFilesDir* (returns a file representing an internal directory for your app) or *getCacheDir* (returns a file representing an internal directory for your app's temporary cache files). Alternatively, you may directly invoke the method *openFileOutput* which opens a file with a given name from the app's file directory, like in the example below.

```
String filename = "settings";
String string = "Hello there!";
FileOutputStream outputStream;

try {
    outputStream = openFileOutput(filename, Context.MODE_PRIVATE);
    outputStream.write(string.getBytes());
    outputStream.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

You can also read about dealing with writing to external storage at <https://developer.android.com/training/basics/data-storage/files.html>.



1.3. Saving to a database

Saving data to a database is ideal for structured data, so Android provides full support for SQLite databases. Any databases you create will be accessible by name to any class in the application, but not outside the application.

The recommended method to create a new SQLite database is to create a subclass of *SQLiteOpenHelper* and override the *onCreate()* method, in which you can execute a SQLite command to create tables in the database. For an example see the tutorial at <https://developer.android.com/guide/topics/data/data-storage.html#db>.



Also, you can use the network connection to store and retrieve data on your own web-based services. To do network operations, use existing classes in the following packages:

- [java.net.*](#)
- [android.net.*](#)
- <https://developer.android.com/training/data-storage/room>

More details and code examples to be found online at <https://developer.android.com/guide/topics/data/data-storage.html>.



2. Google Firebase

Google has introduced a platform for creating rich mobile and web applications that require all the modern elements: server-side storage options, authentication, notifications, messaging, analytics and more. We will rely on Firebase to develop our own project application, and will introduce each required element in time, as we will make use of it. To get started, you can read about setting up Firebase in your Android project here: <https://firebase.google.com/docs/android/setup>



We will rely on the very rich and simple to follow documentation provided by Google. This can be found online at the link and QR code provided just above. This week we will introduce the real-time database storage. It allows storage and synchronization of data with Google's NoSQL cloud database. Data is stored as JSON and synchronized in real-time to every connected client. When you build cross-platform apps with our iOS, Android, and JavaScript SDKs, all of your clients share one Realtime Database instance and automatically receive updates with the newest data. How does it work? Read at <https://firebase.google.com/docs/database/> for more details.



While you might be used to classic SQL databases and design, we will use a different approach, namely NoSQL. Make sure you read a bit about it at <https://en.wikipedia.org/wiki/NoSQL>



2.1. Project setup

1. Create a new project in Android Studio named Smart Wallet and make sure the application id (or fully qualified name) is "com.upt.cti.smartwallet"

2. Login to your own Firebase [console](https://console.firebase.google.com/) (<https://console.firebase.google.com/>) using your Google account, or login to MSA UPT using the credentials provided in class.
3. We will create the project together in class, so you can skip this step. If you use your own Firebase account, then create a new project called Smart Wallet, like in Figure 1.

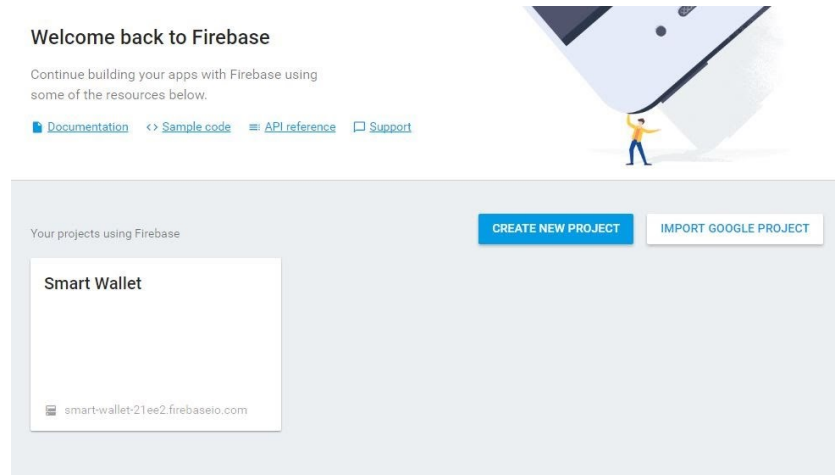


Fig. 1. Firebase welcome screen

4. Next, we will add Firebase to our project in Android Studio. For this step, you will need the fully qualified name of your app. It can be found in the manifest file or in the project build grade file (make sure it's `com.upt.cti.smartwallet` for our assignment). You should see a wizard similar to the one in Figure 2.

Add Firebase to your Android app ×

1 — 2 — 3

Enter app details Copy config file Add to build.gradle

Package name ⓘ Your package name is generally the **applicationId** in your app-level **build.gradle** file

com.upt.cti.smartwallet

Debug signing certificate SHA-1 (optional) ⓘ

00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00

Required for Dynamic Links, Invites, and Google Sign-In support in Auth. Edit SHA-1s in Settings.

CANCEL **ADD APP**

[downloads google-services.json for your app](#)

Fig. 2. Linking the Firebase app through the online console

5. Once the app name is introduced, Firebase will create a configuration file which must be copied inside your project to make the magic work. As Figure 3 details, you need to copy the *google-services.json* file into your app folder. The lab application Smart Wallet has already been configured during the demo session, so please find [here](#) the corresponding json file.

Add Firebase to your Android app

1

2

3


Enter app details

Copy config file

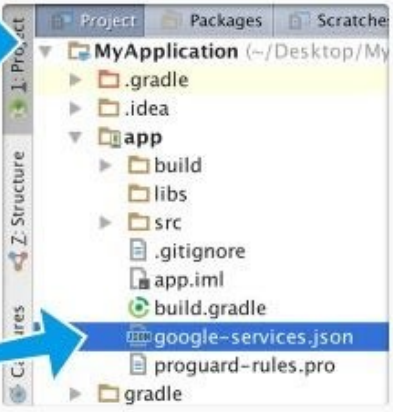
Add to build.gradle

Switch to the **Project** view in Android Studio to see your project root directory.

Move the **google-services.json** file you just downloaded into your Android app module root directory.



google-services.json



Project

MyApplication (~/.Desktop/My)

.gradle

.idea

app

build

libs

src

.gitignore

app.iml

build.gradle

google-services.json

proguard-rules.pro

gradle

Already added the dependencies?
[Skip to the console](#)

CONTINUE

Fig. 3. Adding the Google Services settings file to a project

- Also, you need you need to link this file with your project. It requires two lines of code in your app and project gradle files, and this is done as described in Figure 4.

Add Firebase to your Android app

1

2

3

Enter app details

Copy config file

Add to build.gradle

The Google services plugin for [Gradle](#) loads the `google-services.json` file you just downloaded. Modify your `build.gradle` files to use the plugin.

1. Project-level `build.gradle` (`<project>/build.gradle`):

```
buildscript {  
    dependencies {  
        // Add this line  
        classpath 'com.google.gms:google-services:3.0.0'  
    }  
}
```
2. App-level `build.gradle` (`<project>/<app-module>/build.gradle`):

```
...  
// Add to the bottom of the file  
apply plugin: 'com.google.gms.google-services'  
  
includes Firebase Analytics by default
```
3. Finally, press "Sync now" in the bar that appears in the IDE:

Gradle files have changed since last sync

Sync now

FINISH

Fig. 4. Adding Google Services dependencies to a project Gradle file

7. The only thing that remains on the configuration side is to actually add dependencies to the Firebase SDK. There are multiple libraries available, based on your needs as a developer. These are:
 - `com.google.firebase:firebase-core`: used for analytics
 - `com.google.firebase:firebase-database`: needed for realtime database (we will need this one now!)
 - `com.google.firebase:firebase-storage`: needed for storage (a different functionality from the database, as it offers the possibility to store binary files like video and images; we will use it later)

- com.google.firebase:firebase-crash: used for crash reporting
 - com.google.firebase:firebase-auth: needed for authentication
 - com.google.firebase:firebase-messaging: needed for cloud messaging and notifications
 - com.google.firebase:firebase-config: used for remote config
 - com.google.firebase:firebase-invites: needed for invites and dynamic links
 - com.google.firebase:firebase-ads: used for AdMob (displaying ads)
 - com.google.android.gms:play-services-appindexing: used for app indexing
8. For this assignment we will need two dependencies, so add them in your project gradle file, to the dependencies declaration. If you are using an older version Android API, like for example 23, then you need to adjust the version of Firebase accordingly.

Latest Android API:

```
dependencies{
    compile 'com.google.firebase:firebase-core:9.6.1'
    compile 'com.google.firebase:firebase-database:9.6.1'
}
```

Android API 23:

```
dependencies{
    compile 'com.google.firebase:firebase-core:9.4.0'
    compile 'com.google.firebase:firebase-database:9.4.0'
}
```

9. Our project is now set to work with the real-time database available in our Smart Wallet app.
10. The current data in our database will be a calendar with an entry for every month of the current year holding the income and expenses expressed in our local currency. We have added entries for the first 3 months of the year, as you may see in Figure 5.

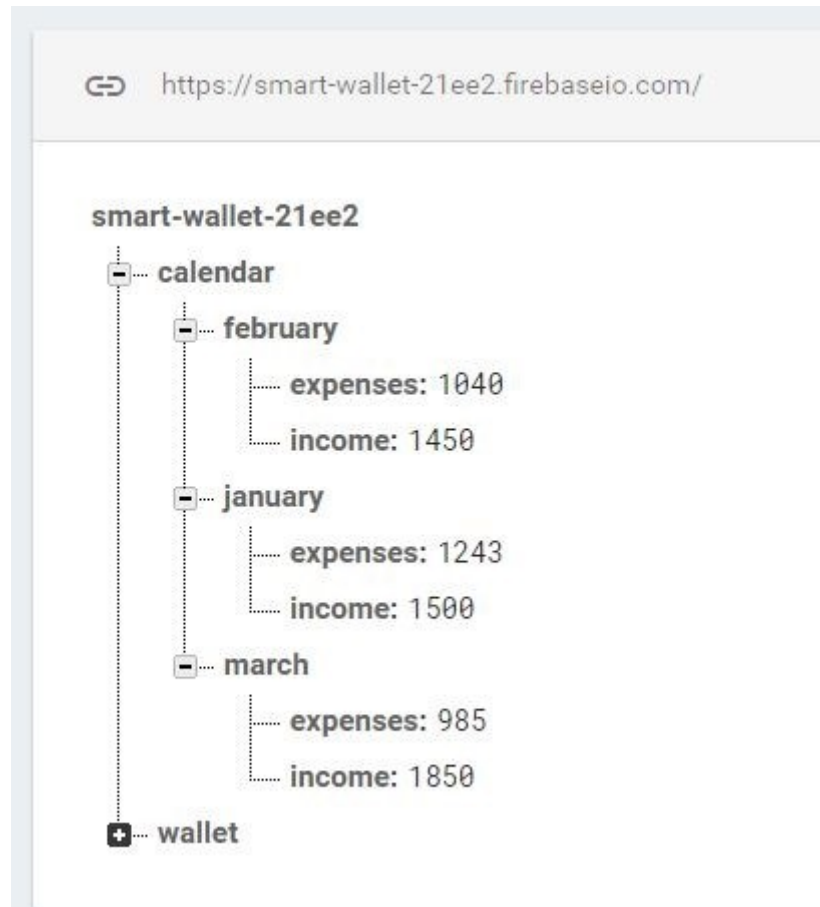


Fig. 5. Example data in the Firebase Realtime Database

3. Task #1

- Based on the data structure defined in Firebase (monthly expenses) we need to create a class that mirrors this structure. For this, create a new package called "model" and add a new Java class named *MonthlyExpenses*. To fully understand the steps and methodology we apply here, please also read the online documentation on [reading](https://firebase.google.com/docs/database/android/read-and-write) data: <https://firebase.google.com/docs/database/android/read-and-write>

The class we add here has two fields named exactly as the fields in Firebase (i.e. *income*, *expenses*). This is important for direct mapping from Json to our Java object at runtime.



```

@IgnoreExtraProperties
public class MonthlyExpenses {

    public String month;
    private float income, expenses;

    public MonthlyExpenses() {
        // Default constructor required for calls to DataSnapshot.getValue()
    }

    public MonthlyExpenses(String month, float income, float expenses) {
        this.month = month;
        this.income = income;
        this.expenses = expenses;
    }

    public String getMonth() {
        return month;
    }

    public float getExpenses() {
        return expenses;
    }

    public float getIncome() {
        return income;
    }
}

```

- Our application will enable us to search a specific month by name, then retrieve and edit the income and expenses values. As such, the layout file of the main activity may look like the one in Figure 6. See code details [here](#).

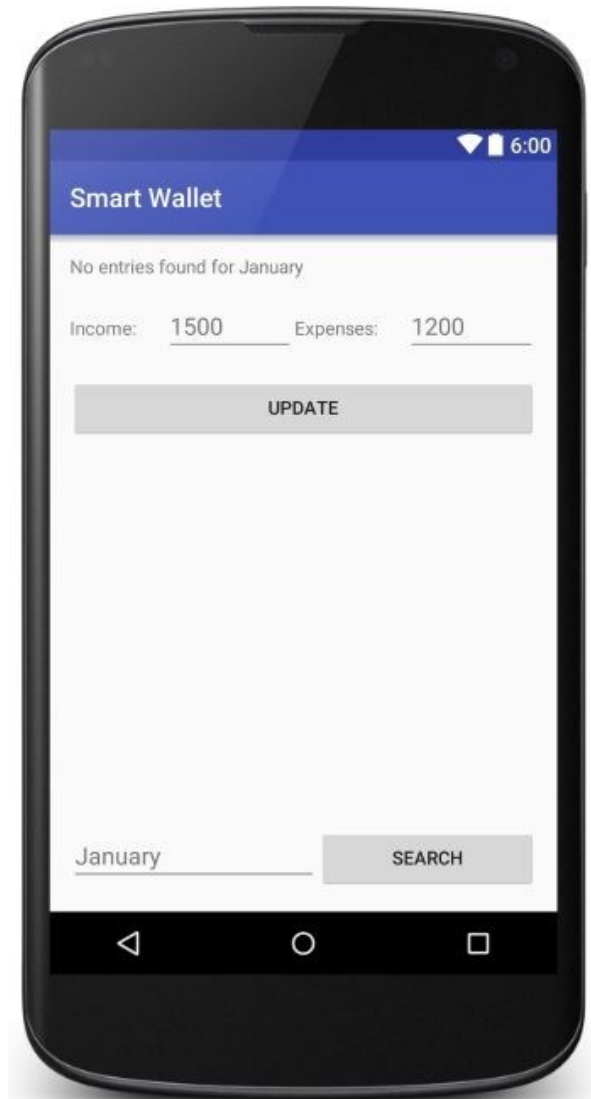


Fig. 6. Suggested user interface for the demo Firebase app

- Programmatically, you need to acquire a reference to the real time database and all UI elements in *onCreate*, as such:

```

// ui
private TextView tStatus;
private EditText eSearch, eIncome, eExpenses;
// firebase
private DatabaseReference databaseReference;

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

    tStatus = (TextView) findViewById(R.id.tStatus);
    eSearch = (EditText) findViewById(R.id.eSearch);
    eIncome = (EditText) findViewById(R.id.eIncome);
    eExpenses = (EditText) findViewById(R.id.eExpenses);

    FirebaseDatabase database = FirebaseDatabase.getInstance();
    databaseReference = database.getReference();
}

```

- Now the tricky part comes from the fact that we do not want to start searches until the user hits the search button. So, we need to handle the user click on *bSearch*. The method `createNewDBListener` is detailed [here](#).

```

public void clicked(View view) {
    switch (view.getId()) {
        case R.id.bSearch:
            if (!eSearch.getText().toString().isEmpty()) {
                // save text to lower case (all our months are stored
                currentMonth = eSearch.getText().toString().toLowerCase();

                tStatus.setText("Searching ...");
                createNewDBListener();
            } else {
                Toast.makeText(this, "Search field may not be empty",
                }
                break;
            case R.id.bUpdate:
                break;
        }
    }
}

```

- At this point, the app should run correctly (in most cases). If you input a search term (like January or February) the app will load the income and expenses values from the online database. However, what happens if your search for April? You should get a crash and a *NullPointerException*. Try to locate the cause and fix this crash.

4. Task #2

- Implement the capability to also update the income and expenses values using the *bUpdate* button. For a reference about writing to Firebase read [here](#), and also ask the lab assistant for hints. Don't forget to check if the input of *eIncome* and *eExpenses* is non empty, is *parsable* to a decimal number and your database reference and current month are non-null.
- Use *shared preferences* to save to current month so that when you relaunch the app, the eSearch edittext will already be filled with the last searched month.
- **Homework:** instead of manually searching using the *bSearch* and *eSearch* views, use a spinner that is automatically updated with the list of months from Firebase (at runtime). The spinner may be used to switch between existing months. Inserting a new month in Firebase should update the spinner at runtime (optional), or after a relaunch of the app. Plus one activity (+1) point for the homework. Homework may only be presented individually, on your smartphone or emulator, with the code running on your PC.

Have a fruitful week.



Bael fruit

The Bael fruit closely resembles a large grapefruit or a large but dull and aging orange. It has dull grayish green color at first which then turns a dirty yellow after ripening. Its hard shell can only be cracked open with a hammer or a machete. On the inside a fibrous yellow pulp with hairy seeds are visible. Taking about 11 months to ripen, this fruit is said to have a taste of marmalade infused with tamarind or lemon indicating a slight hint of tanginess and smells like roses.

<http://www.fruitsinfo.com/bael-fruit.php>