,

Vacation time is here at least, and you’ve just taken the plane that will fly you... **to Greenland**, a trip you've been planning for some time now. In fact, you've already got your winter clothes on! Because you thought ahead, **you bookmarked** several extremely important websites on your cellphone that contain everything you’ll be doing in Greenland. Once you get there, you take out your phone and... No 4G! Not only that, **no Internet access** at all! :waw:

But this misadventure might not need to happen to you... Why? Because you took this course, of course! :D This one will delve into different ways to **store data** in an Android app, accessible **100% offline** (without an Internet connection).

You’ll learn how to **store content** in the "internal" and "external" memory of an Android app, and even how to **organize** structured information and **make it persist** in an **SQLite** database. We’ll also cover **Architecture Components** in order to make our Android apps even more robust and testable.

Naturally, we’ll be developing a fun, useful mini-app to implement all this: **SaveMyTrip**.

**Instructional goals:**

* Create and configure an SQLite database
* Store and retrieve information from a database using SQL language
* Expose content to third-party applications
* Understand how the storage system works in Android
* Implement a permanent architecture in Android

**Prerequisites:**

This course - part of the [Android certification curriculum](https://openclassrooms.com/paths/developpeur-se-d-applications-android) - is mainly for students who have already learned the fundamentals of Java language and taken the courses:

* [Develop Your First Android App](quot;https://openclassrooms.com/courses/developpez-votre-premiere-application-android&quot)
* [Build a Flexible, Adaptable User Interface](quot;https://openclassrooms.com/courses/construisez-une-interface-utilisateur-flexible-et-adaptative&quot)
* [Retrieve and Display Remote Data](quot;https://openclassrooms.com/courses/recuperez-et-affichez-des-donnees-distantes&quot)

## **Store your data in an Android file**

### **Understand different ways of storing data in Android**

#### **Introduction**

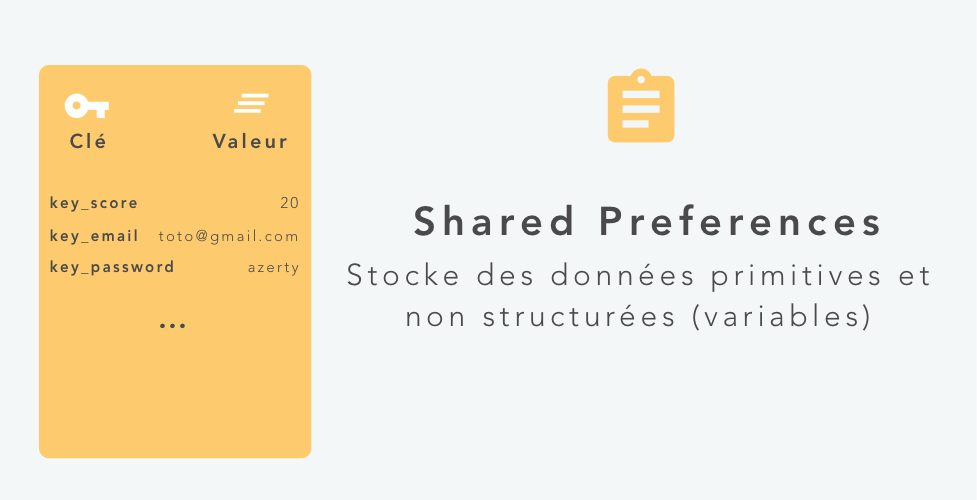
In this course dedicated **to storage in Android**, we’ll study different ways to **save** (or persist) content in our user's phones, 100% offline, meaning without any need for the Internet!

Thus, we’ll discover the different ways Android has to save our app's data on our users’ phones.

#### **1. Shared Preferences**

In [the first course](https://openclassrooms.com/courses/developpez-votre-premiere-application-android/memorisez-les-preferences-du-joueur) of the [Android curriculum](https://openclassrooms.com/paths/developpeur-se-d-applications-android), we already covered [SharedPreferences](https://developer.android.com/training/data-storage/shared-preferences.html), which are a way of storing primitive, simple data: Basically, you store only **variables (values)**, which you then retrieve using **a single identifier (key)**.

This type of storage is called **Key/Value** storage. The variables (and their corresponding keys) will automatically be stored in an XML file directly on the user’s telephone. That file may be defined in PRIVATE mode to allow only our app to access its content, or PUBLIC if you want other apps to be able to access it.

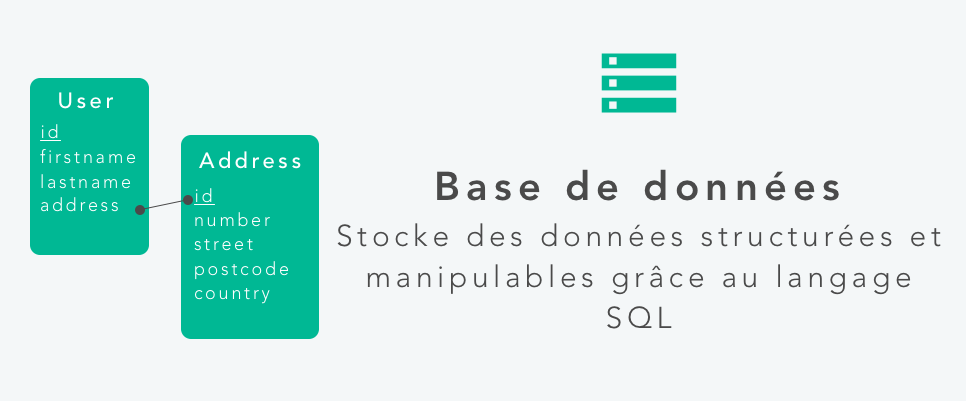
Storage using Shared Preferences

#### **2. Databases**

In order to store **structured** data, Android natively offers you the ability to use an [SQLite](https://wikipedia.org/wiki/SQLite) database. This type of storage will enable you to organize and persist large quantities of structured data, directly on the user’s phone, which can be worked on using SQL language.

Nicolas, one of our talented authors, has created [a special course](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql) dedicated to **SQL databases**. So go ahead and read it to better understand the SQL language.

We’ll be talking about SQLite and its implementation on Android (with Room) in greater detail in the **second part** of this course... :)

Storage using a database

#### **3. Internal and External storage**

In order to support data storage in the form of **files** (photos, videos, PDFs, etc...), Android devices have two storage spaces: The "**Internal"** storage space and the "**External"** storage space.

These fairly common names come from the time when most Android devices had both a physical, non-volatile **internal memory** and a **removable memory** such as an SD card. Today, removable memory is falling out of favor, and many devices now divide their physical storage space into separate partitions: One partition for **internal** memory and another for **external** memory.

So these two storage spaces **still exist**, right? Even if the device doesn’t have a port for removable memory like an SD card?:)

Exactly! :D What's more, there are several differences between these two storage spaces:

|  |  |  |
| --- | --- | --- |
|  | **INTERNAL STORAGE** | **EXTERNAL STORAGE** |
| **Availability** | Files stored in this space will **always** be available. | Files stored in this space will **NOT ALWAYS** be available. That’s because this storage can be removed by the user at any time (SD card or USB key). |
| **Accessibility** | Files stored in this space can **ONLY** be accessed by your application. | Files stored in this space can be accessed by **ANYONE**. You therefore have no control over it. |
| **Lifespan** | When the user uninstalls your app, the files will **automatically be deleted** from this storage space. | When the user uninstalls your app, the files will **NOT** be deleted from this storage space (unless you explicitly defined them to). |

##### **A few words on the INTERNAL storage space...**

By default, the files stored in the *internal storage space* **will be private** and can be accessed **ONLY** by your app. This means they won't be visible to other apps or to the user (unless he or she has a phone with root access... :pirate:).

The *internal storage* space can also be used if you want to save data **temporarily**, in a special folder **dedicated to the cache**. Thus, if your user’s telephone is short on storage space (or if a cleaning program like [Clean Master](https://play.google.com/store/apps/details?id=com.cleanmaster.mguard) has gone through... ;)), the contents of this folder will automatically be erased to free up some space.

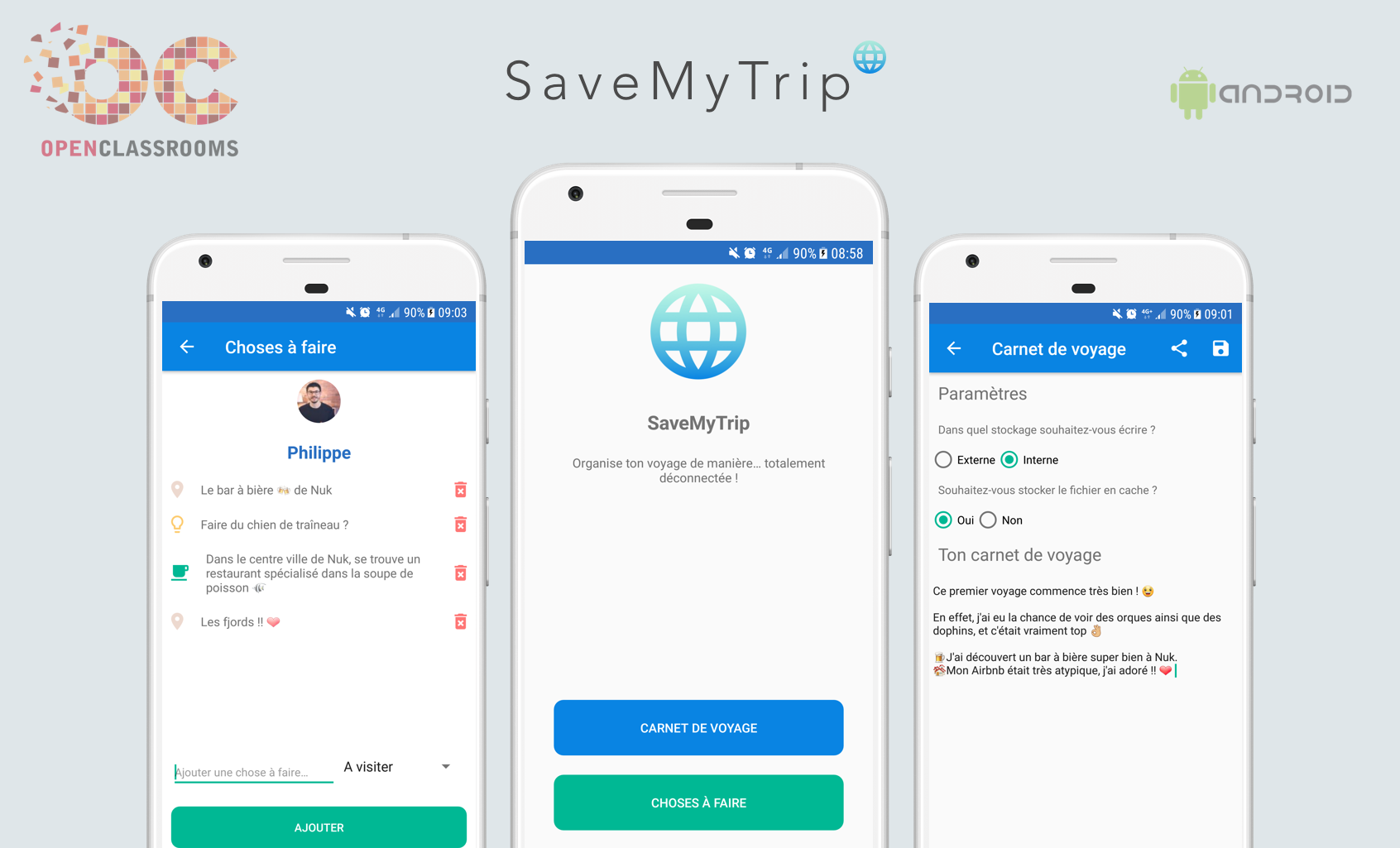
##### **A few words on the EXTERNAL storage space...**

Files stored in the *external storage* space are to be considered **public** and accessible to **everyone**... :waw:

There are also the public folders "Documents", "Downloads" and "Music". So you have **no control** on what files are stored on this storage space. You can simply define them as PUBLIC (the file **will not be** deleted when the user uninstalls your application) or PRIVATE (the file will be deleted when the user uninstalls your application).

Internal and External Storage

#### **Overview of the app SaveMyTrip**

****Overview of the app SaveMyTrip

**SaveMyTrip** is the mini-app that we will develop throughout this course. Its objective is to be able to *easily organize your future trips*, thanks to its fantastic included travel diary as well as its interactive list of things to do, all **accessible 100%off-line, of course**! :p

You can [download this mini-app SaveMyTrip](https://github.com/OpenClassrooms-Student-Center/C7.1-Android-100-offline-app/tree/starter-app) in a nearly blank state, save for a few files:

* **Resource files:** The resource files *colors.xml*, *dimen.xml*, *strings.xml* and *styles.xml file* are already pre-configured.
* **Layout files:** In order to streamline the code found in the chapters, the XML design of the app's three screens was already performed. Feel free to check them out to absorb their info.
* **Classes of activities:** The BaseActivity class, as well as the activity classes MainActivity, TripBookActivity, and TodoListActivity have already been created.
* **Utility class:** The ItemClickSupport class used to more easily manage clicking on the RecyclerView has also been created. We've already seen it in [this course](https://openclassrooms.com/courses/recuperez-et-affichez-des-donnees-distantes/interagir-avec-la-recyclerview).
* **External libraries:** The Glide library is also installed via Gradle.

I have also chosen to organize the structure of the project **by features**, rather than by MVC. But this doesn't change that fact that our architecture will still be MVC... ;)

The purpose of this course is to focus on how we store data in Android, rather than on creating a graphical user interface, which you should be proficient in by now. That's why we're not starting from a totally blank app... :soleil:

And that's all! Open the app SaveMyTrip. The result should look like this:

Glimpse of the blank SaveMyTrip mini-app.

Once again, feel free to browse the already-written base code, so that you understand it a little. As you'll see, it's all simple and was already seen during earlier courses... :)

### **Create a file on the external storage**

Let’s start developing our mini-app **SaveMyTrip**. If you haven’t already done so, [download](https://github.com/PhilippeBoisney/OpenClassrooms---Parcours-Android/tree/cours4-storage.starter-app) and run this mini-application that I pre-made for you... :)

#### **Introduction**

In this section, we will configure the TripBookActivity activity which is responsible for managing the "**travel book**” feature.

The book’s purpose is to allow the user to write the text **in a file** that will be saved in the external or internal storage space of their telephone, depending on the choice they will make via the radio buttons (also called "option boxes"):

* Which storage do you want to write to?
  + **External**
    - What level of privacy do you want?
      * Public: The file will be stored on the *external storage* space in public mode. Therefore it will *not be* *deleted* when the user uninstalls their app.
      * Private: The file will be stored on the *internal storage* space in private mode. Therefore it will *be* *deleted* when the user uninstalls their app.
  + **Internal**
    - Do you want to store the file in the cache?
      * Yes: The file will be stored on the *internal storage* in the directory dedicated to the cache. It may therefore be *deleted at any time*.
      * No: The file will be stored on the internal storage space.

We will begin in this chapter by managing the saving of this file in the **external storage space**: The file can therefore be created either in **private** mode or in **public** mode.

This file will be called **tripBook.txt** and will be placed in a folder called **bookTrip/**. We will therefore create a file (as well as its folder) in both available storage spaces, depending on what our user decides... :)

#### **Save and retrieve data in a file**

##### **Create a utility class**

First things first, we’ll create a singleton object that we’ll call **StorageUtils** and place in the package **utils/** of our application.

**Class utils/StorageUtils.kt:**

**object** StorageUtils {

}

**Explanations:** This object will be responsible for *saving* and *retrieving* data entered into our file tripBook.txt. This will avoid **overloading** our TripBookActivity activity and most importantly will allow us to **re-use** its code in other activities, if need be.

##### **Create an access path to file**

In order to save the text written by our user, a destination file must first be defined. So let’s create the method createOrGetFile() in our class **StorageUtils.java**.

**Class utils/StorageUtils.kt:**

private fun createOrGetFile(

destination: File, // e.g., /storage/emulated/0/Android/data/

fileName: String, // e.g., tripBook.txt

folderName: String) // e.g., bookTrip

: File {

val folder = File(destination, folderName)

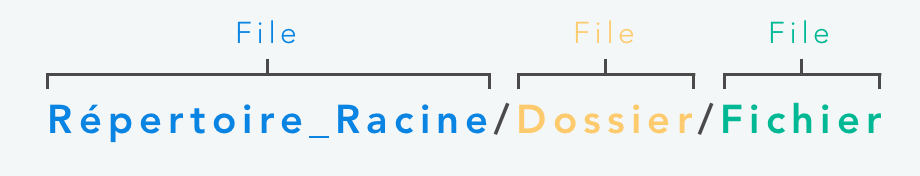
// file path = /storage/emulated/0/Android/data/bookTrip/tripBook.txt

return File(folder, fileName)

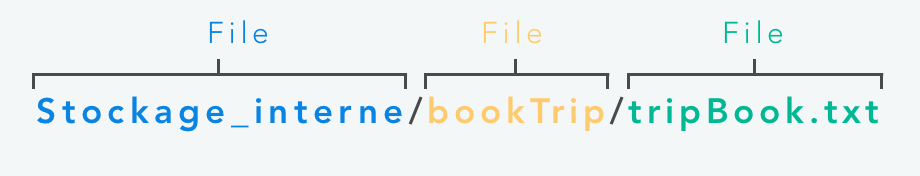
}

**Explanations:** This method will be called to *create* or *retrieve* a file.

The class [File](https://developer.android.com/reference/kotlin/java/io/File.html) is a little misleading, because you might think it’s a representation of a file, but this is not quite true! It actually represents **a path to a file**, which will be used later to save or retrieve a data stream (characters, bytes, etc...).

Example path structure

In our case, we will define a path to a **folder containing a file** (in this case **bookTrip/tripBook.txt**), from a **root destination** specified in a parameter of the method. Don’t worry, you’ll understand it more later... :)

Example use for SaveMyTrip

##### **Read and write data from a file**

Now that we’ve seen how to **create** and **retrieve** a file from a path, it would be good to be able to **read** and **write** inside it. To do this, add the following lines...

**Excerpt from StorageUtils.kt:**

object StorageUtils {

**…**

// ----------------------------------  
 // READ & WRITE STORAGE  
 // ----------------------------------  
 private fun readFile(context: Context, file: File): String {

val sb = StringBuilder()

if (file.exists()) {

try {

val bufferedReader = file.bufferedReader();

bufferedReader.useLines { lines ->

lines.forEach {

sb.append(it)

sb.append("\n")

}

}

} catch (e: IOException) {

Toast.makeText(context, context.getString(R.string.error\_happened), Toast.LENGTH\_LONG).show()

}

}

return sb.toString()

}   
  
 // ---  
  
 private fun writeFile(context: Context, text: String, file: File) {

try {

file.parentFile.mkdirs()

file.bufferedWriter().use {

out -> out.write(text)

}

} catch (e: IOException) {

Toast.makeText(context, context.getString(R.string.error\_happened), Toast.LENGTH\_LONG).show()

return

}

Toast.makeText(context, context.getString(R.string.saved), Toast.LENGTH\_LONG).show()

}

}

**Explanations:** Here we’ve added two general methods: readFile() and writeFile() .

The method readFile() will allow us to **read the contents** of a file passed as a parameter. It first creates a [StringBuilder](https://developer.android.com/reference/kotlin/java/lang/StringBuilder) to accumulate the contents of the file as it is read. Then, it verifies that the file exists before obtaining a [BufferedReader](https://developer.android.com/reference/kotlin/java/io/BufferedReader.html) from the [File](https://developer.android.com/reference/kotlin/java/io/File) to read the data stream efficiently using its memory buffer. Within the useLines() block the contents of the file are read **line by line** appending each line to the [StringBuilder](https://developer.android.com/reference/kotlin/java/lang/StringBuilder). Upon completion the method returns a string that contains the entire contents of the file.

Additionally, the method writeFile() will allow us **to write the text** into a file. This method creates any folder(s) needed in the path that don’t already exist (in our case, **bookTrip/**) using the method [mkdirs()](https://developer.android.com/reference/java/io/File.html%23mkdirs()). Next, it obtains a [BufferedWriter](https://developer.android.com/reference/kotlin/java/io/BufferedWriter.html) to open a data stream to the file and writes the contents of the file to it in the use() block.

These two methods are fairly **"rough work"**, don’t spend too much time trying to understand them in detail. Just try to **absorb the logic** of the process... :)

##### **Manipulate storage spaces**

Now that our general methods of writing and reading are ready, we will be able to manipulate our **external** and **internal** storage with them. To do so, add the following methods to your StorageUtils.java class.

**Excerpt from StorageUtils.kt:**

object StorageUtils {

fun getTextFromStorage(rootDestination: File,

context: Context,

fileName: String,

folderName: String): String? {

val file = createOrGetFile(rootDestination, fileName, folderName)

return readFile(context, file)

}

fun setTextInStorage(rootDestination: File,

context: Context,

fileName: String,

folderName: String,

text: String) {

val file = createOrGetFile(rootDestination, fileName, folderName)

writeFile(context, text, file)

}

*// ----------------------------------  
 // EXTERNAL STORAGE  
 // ----------------------------------* **fun** isExternalStorageWritable(): Boolean {

**val** state = Environment.getExternalStorageState()  
 **return** Environment.*MEDIA\_MOUNTED* == state

}  
  
 **fun** isExternalStorageReadable(): Boolean {

**val** state = Environment.getExternalStorageState()  
 **return** Environment.*MEDIA\_MOUNTED* == state ||

Environment.*MEDIA\_MOUNTED\_READ\_ONLY* == state

}

**Explanations:** Here we’ve added several methods: The last **two** are used to check if the **external** storage is *available* and if it can be *read* (isExternalStorageReadable ) or *written to* (isExternalStorageWritable ).

Then we created two other methods to write and read text in a file **located in a storage space defined in a parameter**... by reusing the methods writeFile and createOrGetFile that we created earlier! :D

OK, but I do not understand, does that means that we will write to **the same file** in the **same place** each time? o_O

Nope! Take a closer look at these methods. We pass a **destination ROOT directory (**the parameterrootDestination **)** to the method createOrGetFile , which allows us to **create or retrieve a file** (tripBook.txt) and **its folder** (bookTrip/) from that root directory!

This amazing root directory will actually represent **the root of the storage space** in which we want to save our file!

Now that everything’s ready, we call all of these methods in our controller, **TripBookActivity**.

#### **Updating the activity**

Now let's edit our TripBookActivity activity in order to call these last two methods when a user takes an action.

**Excerpt from TripBookActivity.kt:**

class TripBookActivity : BaseActivity() {  
  
 *// 1 - FILE MANAGEMENT* private val FILENAME = "tripBook.txt"  
 private val FOLDERNAME = "bookTrip"

...

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

...

// 2 - Read from storage when starting

this.readFromStorage()

}

// ----------------------------------

// ACTIONS

// ----------------------------------

fun onClickRadioButton(button: CompoundButton, isChecked: Boolean) {

if (isChecked) {

...

}

// 3 - Read from storage after user clicks on radio buttons

this.readFromStorage()

}

override fun onOptionsItemSelected(item: MenuItem): Boolean {

when (item.itemId) {

...

R.id.action\_save -> {

// 4 – Save

this.save()

return true

}

}

...

}

// 5 - Save after user clicked on button

private fun save() {

if (radioButtonExternal.isChecked) {

writeExternalStorage() //Save on external storage

} else {

//TODO: Save on internal storage

}

}

// ----------------------------------

// UTILS - STORAGE

// ----------------------------------

// 6 - Read from storage

private fun readFromStorage() {  
  
 if (radioButtonExternal.isChecked) {  
  
 if (StorageUtils.isExternalStorageReadable()) {

// EXTERNAL

if (radioButtonExternalPublic.isChecked) {

// External - Public  
 editText.setText(StorageUtils.getTextFromStorage(

Environment.getExternalStoragePublicDirectory(

Environment.*DIRECTORY\_DOCUMENTS*), **this**, **FILENAME**, **FOLDERNAME**))

} **else** {

*// External - Privatex* **editText**.setText(StorageUtils.getTextFromStorage(

getExternalFilesDir(Environment.*DIRECTORY\_DOCUMENTS*)!!, **this**, **FILENAME**, **FOLDERNAME**))  
 }

}  
  
 } **else** {  
  
 *// TODO : READ FROM INTERNAL STORAGE*  
 }

// 7 - Write external storage

private fun writeExternalStorage() {

if (StorageUtils.isExternalStorageWritable()) {

if (radioButtonExternalPublic.isChecked) {

StorageUtils.setTextInStorage(

Environment.getExternalStoragePublicDirectory(

Environment.DIRECTORY\_DOCUMENTS),

this,

FILENAME,

FOLDERNAME,

this.editText.text.toString())

} else {

StorageUtils.setTextInStorage(

getExternalFilesDir(Environment.DIRECTORY\_DOCUMENTS),

this,

FILENAME,

FOLDERNAME,

this.editText.text.toString())

}

} else {

Toast.makeText(this, getString(R.string.external\_storage\_impossible\_create\_file),

Toast.LENGTH\_LONG).show()

}

}

}

**Explanations:** In this activity, we've called the two public methods of object StorageUtils which allow us to write or read a file from a root directory.

And this root directory will change depending on what the user chooses? :soleil:

Indeed! First we declared some variables **(1).** FILENAME **is the name of the file (tripBook.txt)** in which we wish to store text, and FOLDERNAME is **the name of the folder** (**bookTrip/**) that will contain this file.

At the end of our onCreate method we call the method readFromStorage so that every time the activity is launched **(2)** we get read the latest file contents.

All of the radio buttons have click listeners that call the onClickRadioButton method which now has a call to readFromStorage **(3)** as well.

In the onOptionsItemSelected method **(4)** we are responding when the user presses the "Save" button on the Toolbar by calling the save method **(5).**

Next, we created a method **(6)**, readFromStorage , in order to read the contents of the file depending on what the user chooses (via the radio buttons). We did the same thing **(7)** in the method writeExternalStorage , but this time by writing of course... :)

Additionally, as you may have noticed, these are the same two methods that call the methods created previously in the class **StorageUtils** (getTextFromStorage and setTextInStorage), with of course, **different root directories**:

* Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY\_DOCUMENTS): We will make it possible to generate a path (File) to the "Documents" directory of the **external** storage space, in **public** mode.
* getExternalFilesDir(Environment.DIRECTORY\_DOCUMENTS) : We will make it possible to generate a path (File) to the "Documents" directory of the **external** storage space, in **private** mode.

At this stage, if you launch the application, it will return error messages and the user will be unable to save any content... :waw: I’ll let you figure out why!

#### **Ask permission**

That's correct! As you’ve guessed, **saving** and **reading** a file **on the external storage space** of your users’ phone requires special permissions. For this reason, we must first declare these permissions in our application.

In order to ensure the compatibility of devices whose version of Android is version 6 or above, we will use the library [EasyPermissions](https://github.com/googlesamples/easypermissions) that we already saw [in this course](https://openclassrooms.com/courses/creez-un-backend-scalable-et-performant-sur-firebase/recuperez-une-image-locale-sur-le-telephone).

dependencies {

...

//EASY PERMISSIONS

implementation 'pub.devrel:easypermissions:1.1.1'

}

**Explanations:** Here, the library EasyPermissions is installed, which will enable us to facilitate our request for permissions on Android versions 6 and above.

Next, we will declare the permissions in the manifest of our Android app (for versions at or below 5.1.1):

<manifest

xmlns:android="http://schemas.android.com/apk/res/android"

package="com.openclassrooms.savemytrip">

<!-- ENABLE PERMISSIONS ABOUT EXTERNAL STORAGE ACCESS -->

<uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE" />

...

</manifest>

**Explanations:** We declare the permissions needed to **write** and **read** the content into Android’s external storage, using the permission [WRITE\_EXTERNAL\_STORAGE](https://developer.android.com/reference/android/Manifest.permission.html%23WRITE_EXTERNAL_STORAGE).

OK, but why didn’t you also add the permission [READ\_EXTERNAL\_STORAGE](https://developer.android.com/reference/android/Manifest.permission.html%23READ_EXTERNAL_STORAGE)? :o

Simply because it is **implicitly approved** at the same time the WRITE\_EXTERNAL\_STORAGE permission is! :) Now let’s edit our activity in order to configure EasyPermissions.

**Excerpt from TripBookActivity.kt:**

import static android.Manifest.permission.WRITE\_EXTERNAL\_STORAGE;

*// 1 – PERMISSIONS MANAGEMENT*

**private const val** *RC\_STORAGE\_WRITE\_PERMS* = 100  
  
**class** TripBookActivity : BaseActivity() {

...

// 2 - After permission granted or refused

override fun onRequestPermissionsResult(requestCode: Int,

permissions: Array<String>, grantResults: IntArray) {

super.onRequestPermissionsResult(requestCode, permissions, grantResults)

EasyPermissions.onRequestPermissionsResult(requestCode, permissions, grantResults, this)

}

...

// ----------------------------------

// UTILS - STORAGE

// ----------------------------------

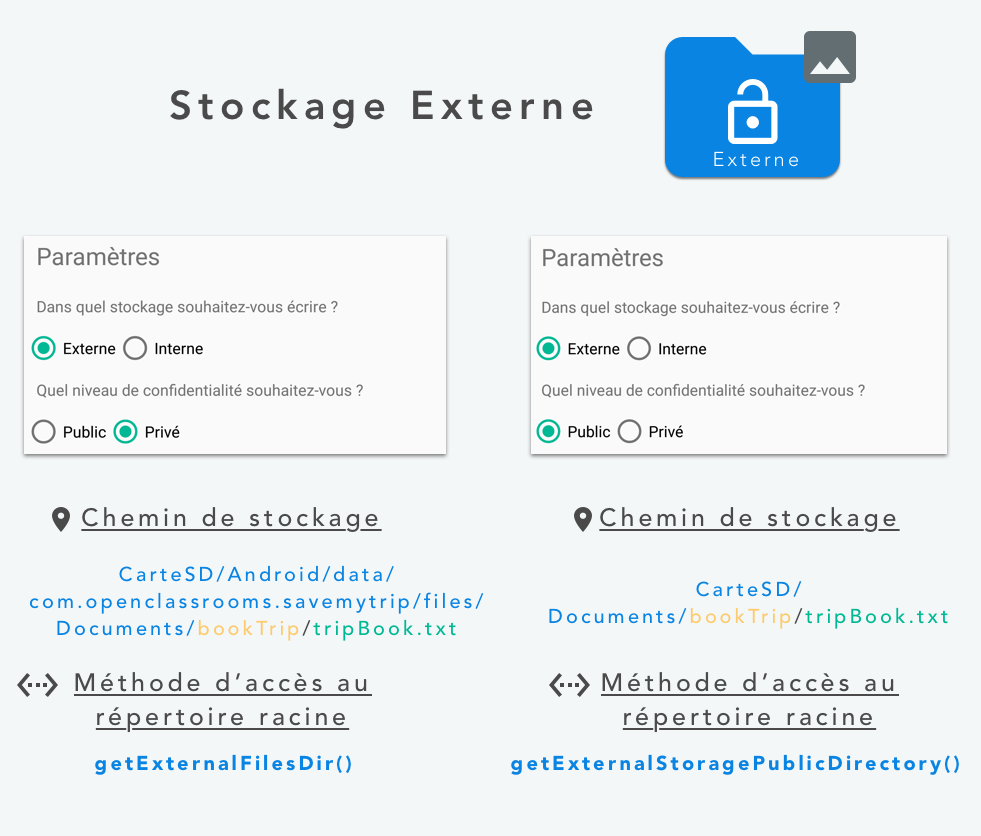
@AfterPermissionGranted(*RC\_STORAGE\_WRITE\_PERMS*)

private fun readFromStorage() {  
  
 // CHECK PERMISSION  
 if (!EasyPermissions.hasPermissions(this, WRITE\_EXTERNAL\_STORAGE)) {  
 EasyPermissions.requestPermissions(this, getString(R.string.title\_permission), RC\_STORAGE\_WRITE\_PERMS, WRITE\_EXTERNAL\_STORAGE)  
 return  
 }

...  
   
 }

**Explanations:** We configure EasyPermissions here so that it prompts the user to accept the WRITE\_EXTERNAL\_STORAGE permission as soon as the activity launches (and tries to read the file from the storage space). Feel free to [reread the chapter of this course](https://openclassrooms.com/courses/creez-un-backend-scalable-et-performant-sur-firebase/recuperez-une-image-locale-sur-le-telephone) for more explanations on this library.

Now run your Android app... You should be able to **save** and **read** the contents of the file without any problem. Also consider **using the file manager** on your phone to view the created files. :)

Saving to external storage

Take the time to **read** and **reread** this code in order to familiarize yourself with all the concepts it refers to. And don't forget the golden rule of any good developer, **Practice Makes Perfect**! Good luck... :ange:

### **Create a file in internal storage**

Now that we know how to properly save content to external storage, we will allow our users to save their text to their phone's **internal storage**, in order to protect it even more... :)

Now I imagine that we will still need to **ask the user’s permission** to use their internal storage, right?

Actually, we don't! When you save information on the internal storage of your users’ phone, you do not need their permission since this space is allocated only to your app... ;)

So in this chapter, we will allow our users to save their text (and consequently the file **tripBook.txt)** in the internal memory of their phone:

* Either in the app's **internal** storage.
* Or in the **dedicated cache** of the app's **internal** storage.

#### **Updating the activity**

Now let's get really serious! Since we created a rather general class utility in the previous chapter, **StorageUtils**, we’ll be reusing it here (but without editing it, obviously!).

Let’s update our activity in order to call the appropriate methods when the user clicks on the radio buttons dedicated to storage in internal memory.

**Excerpt from TripBookActivity.kt :**

**class** TripBookActivity : BaseActivity() {

...

// --------------------

// ACTIONS

// --------------------

...

**private fun** save() {  
  
 **if** (**radioButtonExternal**.*isChecked*) {  
 writeExternalStorage() *//Save to external storage* } **else** {

// 3 - Save on internal storage  
 writeInternalStorage() *//Save to internal storage* }

}

// ----------------------------------

// UTILS - STORAGE

// ----------------------------------

@AfterPermissionGranted(RC\_STORAGE\_WRITE\_PERMS)

**private fun** readFromStorage() {

...

if (radioButtonExternal.*isChecked*) {

...

} else {

// 2 - Read from internal storage

if (radioButtonInternalVolatile.isChecked) {

// Cache

editText.setText(StorageUtils.getTextFromStorage(cacheDir, this, FILENAME, FOLDERNAME))

} else {

// Normal

editText.setText(StorageUtils.getTextFromStorage(filesDir, this, FILENAME, FOLDERNAME))

}

}

}

...

// 1 - Write internal storage

private fun writeInternalStorage() {

if (radioButtonInternalVolatile.isChecked) {

StorageUtils.setTextInStorage(cacheDir,

this,

FILENAME,

FOLDERNAME,

editText.text.toString())

} else {

StorageUtils.setTextInStorage(filesDir,

this,

FILENAME,

FOLDERNAME,

editText.text.toString())

}

}

}

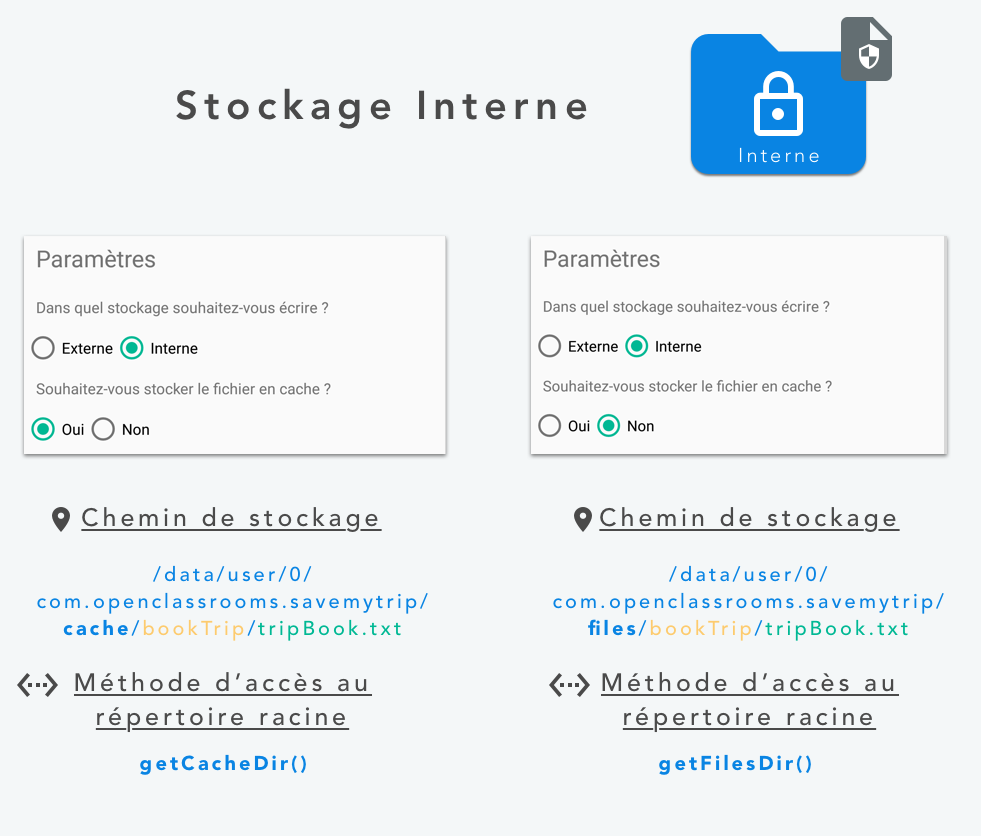
**Explanations:** We've added a method **(1)** named writeInternalStorage dedicated to writing in our file in internal storage, which will be called by the save method **(3)** when the user clicks on the "Save" button on the Toolbar. We've also edited **(2)** the method readFromStorage in order to read the file **from internal storage** whenever the activity is run or when the user clicks on the radio buttons.

And what’s more, all this should remind you of something... ;) Yes, we used the same approach to write and read to our external storage in the previous chapter, by calling the methods getTextFromStorage and setTextInStorage, **changing only** the parameter corresponding to the **root directory** to match that of the internal storage:

* filesDir : This property contains the path (file) to the root of the **internal** storage space for your application.
* cacheDir : This property contains the path (file) to the root of the **cache** of your app's **internal** storage space.

Now run your app and try saving text in the **internal storage space**! :)

Note, **you will not be able to see** the two files created, simply because you do not have access as a user (unless your phone is rooted). That’s what **internal** storage is good for! ;)

Saving to internal storage

Since this chapter is shorter, take the time to **read** and **reread** your code, in order to understand the logic behind it. And don't forget the golden rule of any good developer, **Practice Makes Perfect**! Good luck... :ange:

### 

### **Share a file using FileProvider**

Your users are delighted to be able to write everything that enters their heads in their travel book! However, they would now like to be able to **share this travel book**... and more particularly the one that is located in the **internal** storage, which is theoretically inaccessible to the user.

#### **Discover the FileProvider**

In order to **expose the content** in a secure manner on Android to third-party apps (such as providing a file containing text... ;)), it is very common to use the class [ContentProvider](https://developer.android.com/reference/android/content/ContentProvider.html). That class will generate a secure [URI](https://developer.android.com/reference/java/net/URI.html), allowing anyone who possesses it to access the resource that you’ve shared.

The class [FileProvider](https://developer.android.com/reference/android/support/v4/content/FileProvider.html) is a child class inheriting from ContentProvider, allowing you to securely expose a resource of the type [File](https://developer.android.com/reference/kotlin/java/io/File.html), which is of course a file... :) In our case, we’ll use this class in order to share the file **tripBook.txt** located in our internal memory.

#### **Define a FileProvider**

In order to begin using a FileProvider, we must declare an instance in the manifest of our application:

**Excerpt from AndroidManifest.xml:**

<manifest

xmlns:android="http://schemas.android.com/apk/res/android"

package="com.openclassrooms.savemytrip">

...

<application

...>

...

<!-- FileProvider - Expose File -->

<provider

android:name="android.support.v4.content.FileProvider"

android:authorities="com.openclassrooms.savemytripkt.fileprovider"

android:exported="false"

android:grantUriPermissions="true">

<meta-data

android:name="android.support.FILE\_PROVIDER\_PATHS"

android:resource="@xml/provider\_paths" />

</provider>

...

</application>

</manifest>

**Explanations:** Here we've declared our FileProvider directly in XML in our application's manifest.

The declaration is made using the [provider](https://developer.android.com/guide/topics/manifest/provider-element.html) tag, which allows us to declare a ContentProvider:

* android:name : Corresponds to the name of the class that implements the ContentProvider, here FileProvider of the package *android.support.v4*.
* android:authorities : Used to uniquely identify the authority that is exposing the data.
* android:exported : Used to define whether the content provider can be accessed by applications other than ours. In our case, we just want ours to be able to share content, but without other apps being able to access it.
* android:grantUriPermissions : Used to allow or disallow actions that generally require special permissions, like the rights to read or write on a storage space, for example.

We've also defined, using the **meta-data** tags, the location that we wish to expose. To do this, create in the folder res/ the sub-folder xml/ and place the following file into it:

**File res/xml/provider\_paths.xml:**

<?xml version="1.0" encoding="utf-8"?>

<paths>

<files-path name="BookTrip" path="bookTrip/"/>

</paths>

**Explanations:** We indicate here the folder in which we previously placed our **tripBook.txt** file and that we wish to expose.

And that’s it, the rest will happen in our TripBookActivity controller!

#### **Share a file**

First, we are going to add to our object **StorageUtils** a method for retrieving a file from a defined location, in order to thereafter retrieve our **tripBook.txt** file from **internal** storage.

**Excerpt from StorageUtils.kt:**

object StorageUtils {

...

fun getFileFromStorage(rootDestination: File, context: Context,

fileName: String, folderName: String): File? {

try {

return createOrGetFile(rootDestination, fileName, folderName)

}

catch(e: NullPointerException) {

Toast.makeText(context, context.getString(R.string.error\_happened),

Toast.LENGTH\_LONG).show()

}

return null

}

...

}

**Explanations:** Nothing too complicated here, we simply retrieve a file from a root directory passed as a parameter ( rootDestination ).

Now, let’s edit our TripBookActivity activity in order to share the file.

**Excerpt from TripBookActivity.kt:**

class TripBookActivity : BaseActivity() {

...

// 1 - Define the authority of the FileProvider

private val AUTHORITY = "com.openclassrooms.savemytripkt.fileprovider"

...

override fun onOptionsItemSelected(item: MenuItem): Boolean {

when (item.itemId) {

R.id.action\_share -> {

this.shareFile()

return true

}

...

}

return super.onOptionsItemSelected(item)

}

...

// ----------------------------------

// SHARE FILE

// ----------------------------------

// 2 - Share the internal file

private fun shareFile() {

val internalFile = StorageUtils.getFileFromStorage(filesDir, this, FILENAME, FOLDERNAME)

val contentUri = FileProvider.getUriForFile(applicationContext, AUTHORITY, internalFile!!)

val sharingIntent = Intent(Intent.ACTION\_SEND)

sharingIntent.type = "text/\*"

sharingIntent.putExtra(Intent.EXTRA\_STREAM, contentUri)

startActivity(Intent.createChooser(sharingIntent,

getString(R.string.trip\_book\_share)))

}

**Explanations:** First, we created a variable **(1)** named AUTHORITY containing the identifier of our authority (the one that was declared in our manifesto). Then, we created a method **(2)** called shareFile which will be called as soon as the user clicks on the "Share" button **(3)** on the Toolbar.

Let’s talk a little bit more about the method shareFile . It first retrieves the file located in our internal storage space using the method previously created in our **StorageUtils** object. Next. using the method FileProvider.getUriForFile , we are going to **generate a secure URI** (therefore a unique identifier) pointing to our file.

Now that this URI has been generated, we will be able to create a sharing intent ([Intent.ACTION\_SEND](https://developer.android.com/reference/android/content/Intent.html%23ACTION_SEND)), using that URI to share our file! :)

And that's all! Run your app and click on the "Share" button. You should now be able to share this file, even though it is stored in the internal memory of your application... ;)

Share a file from the internal memory

Now that you are comfortable with **storing files** on Android’s storage spaces (internal and external), the next part of this course will focus on the persistence of structured data in a SQLite database.

## **Store data in a structured way using a SQLite database**

### **Discover the SQLite Room database**

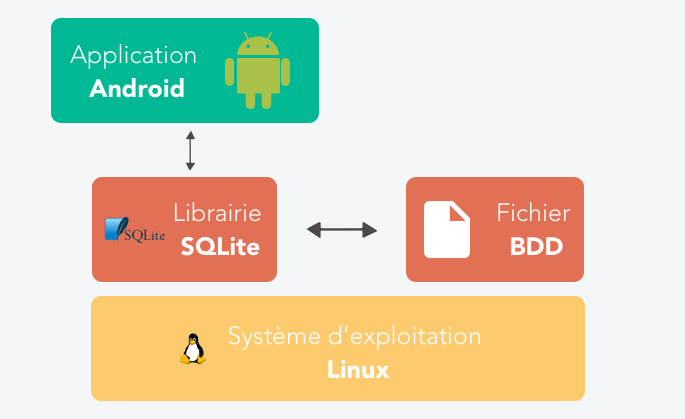
In this section, we will focus on how to save structured data on our users’ phones.

It’s easy, we did it in the last section! Open a file, and store our structured data inside, right? :)

Well, not really! ;) When I say structured data, I mean data that has **strong relationships** between them. Generally, this type of data is stored in a **relational database**. In Android, we have the ability to store this data in a database called "**SQLite"**.

If you aren't familiar with SQL, I strongly advise you to read [the course](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql) by Nicolas, one of our talented authors.

Back to SQLite. It’s a **relational database engine** that can be fully manipulated with SQL language. Unlike traditional database servers such as [MySQL](https://wikipedia.org/wiki/MySQL), [PostgreSQL](https://wikipedia.org/wiki/PostgreSQL) or [Microsoft SQL Server](https://wikipedia.org/wiki/Microsoft_SQL_Server), the entire database is **stored in a file** (not on a remote server).

How SQLite works

Because it takes up so little space ([the source code](https://www.sqlite.org/download.html) of SQLite is less than 2 MB), SQLite can be found in a lot of software such as web browsers, [embedded systems](https://fr.wikipedia.org/wiki/Système_embarqué), and operating systems like Android... :)

Historically in Android, manipulating an SQLite database was... how to put it... quite [complicated](https://developer.android.com/training/data-storage/sqlite.html)!

First, because SQL queries were **“hard-coded”** in static String variables...

private static final String SQL\_CREATE\_ENTRIES =

"CREATE TABLE " + FeedEntry.TABLE\_NAME + " (" +

FeedEntry.\_ID + " INTEGER PRIMARY KEY," +

FeedEntry.COLUMN\_NAME\_TITLE + " TEXT," +

FeedEntry.COLUMN\_NAME\_SUBTITLE + " TEXT)";

The disadvantage of this method is that there is **no option for verification** by your IDE (Android Studio) at the **time of the compilation**. This means that if there is an error, you'll ONLY see it when your app crashes... :waw:

Secondly, because you natively needed a **lot of code** to transform the result of a query into a Java object. Now, ORMs have begun to appear to compensate for these problems!

Uh, what’s an ORM?

Well, an **object-relational mapping** is a programming technique that allows developers to deliberately create **the illusion of manipulating a object-oriented database**, when in reality, we're working with a relational database. ORM will allow us to create an **extra abstraction layer**, so that we can manipulate a relational database (like SQLite!) with objects. :)

There are many open source ORM projects for Android,such as [greenDAO](https://github.com/greenrobot/greenDAO), [OrmLite](https://github.com/j256/ormlite-android), etc with the goal of more easily manipulating a SQLite database.

There is also the ORM [realm.io](https://realm.io/) which is also very popular in Android... ;)

All this tinkering led Android to take things into its own hands, releasing its own ORM, [Room](https://developer.android.com/training/data-storage/room/index.html) ! :D

This release appeared at the same time as the release of the [Architecture Components](https://developer.android.com/topic/libraries/architecture/index.html), a sort of guide to **best practices** and **libraries** for Android Developers in order to improve the quality of their apps. I’ll talk about it more detail in the next part of this course.

In this part, we’ll be using Room to facilitate the storage of our structured data in Android’s SQLite database.

To do so, we’ll install Room in our SaveMyTrip Android project, editing our build.gradle file as usual.

**Excerpt of build.gradle:**

dependencies {

...

//ROOM

implementation 'android.arch.persistence.room:runtime:1.1.1'

annotationProcessor "android.arch.persistence.room:compiler:1.1.1"

//VIEW MODEL & LIVE DATA

implementation "android.arch.lifecycle:extensions:1.1.1"

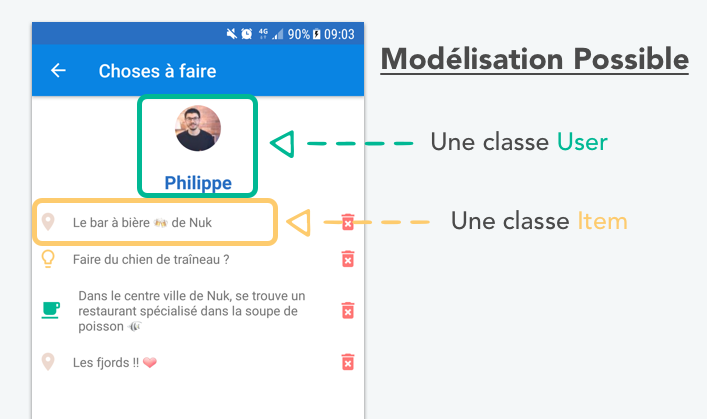
}

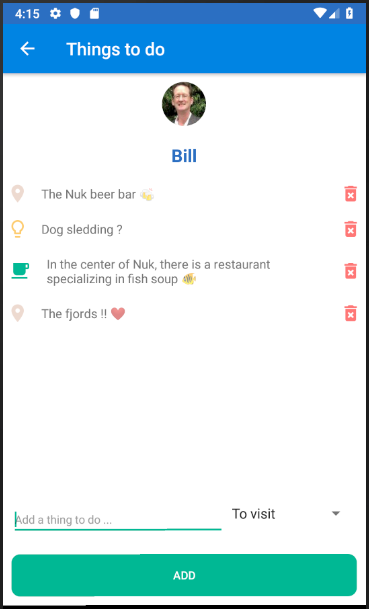
Sync your project and that's it! In the next chapter, we’ll go into a little more detail about Room by creating our first entities.

### **Define a data structure with entities**

Now that Room (our ORM) is installed, we’ll be able to begin representing our data in the form of models (via simple [POJO](https://wikipedia.org/wiki/Plain_old_Java_object) classes ). The goal will be to **model the feature** "*list of things to do*" in our **TodoListActivity** screen.

For the sake of clarity, we’ll go back to this screen in order to better visualize our needs and especially the models that we will have to create.





Example model

In our case, I've chosen to model this feature in the following way:

* The class User : Will represent the user currently logged in to our application. That user **will have** a name and a profile picture.
* The class Item : Will represent a **thing to do**. It will have a **title** and a **category,** and will have the option to be **marked** as **"done".** It will also keep the created **user login** in memory.

Now, let’s create the corresponding classes in Kotlin... :) To do this, create a new package called **models/** then add the classes **Item.kt** and **User.kt** to it.

**Class Item.kt :**

class Item {  
   
 var id: Long = 0,

var text: String = "",

var category: Int = 0,

var selected: Boolean = false,

var userId: Long = 0

}

**Class User.kt:**

class User {

var id: Long = 0;

var username: String = "";

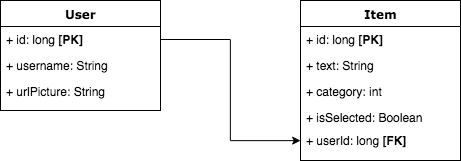
var urlPicture: String = "";

}

**Explanations:** For now, we’ve created these two objects are quite basic but we will add details soon.

You're right, these classes models are quite simple, but how do we save them in a database? :euh:

To help us, I first created a [Physical Data Model](https://openclassrooms.com/courses/faites-une-base-de-donnees-avec-uml/modelisez-le-domaine-fonctionnel%23/id/r-4457872) (PDM) in order to better represent the relationships between these objects, which will later become... **tables**! :)



The physical data model for our app SaveMyTrip

Here we have two tables, corresponding to our two previous models:

* **The table** User**:** 
  + **Primary Key (PK):** The field id representing our user’s unique identifier.
* **The table** Item:
  + **Primary Key (PK):** The field id representing the unique identifier of a "thing to do".
  + **Foreign Key (FK):** The field userId representing the unique identifier of the user who created the "thing to do".

Now that we have a slightly more advanced representation of our database's structure, we will edit our two classes of models in order to **define** them as **tables** in the eyes of Room.

**Excerpt from User.kt:**

@Entity

class User {

@PrimaryKey

var id: Long = 0,

...

}

**Explanations:** To make our lives easier, Room offers us a lot of annotations in order to facilitate its configuration. Thus, here we've defined our class "user" **as a table** using the annotation [@Entity](https://developer.android.com/reference/android/arch/persistence/room/Entity.html).

A table must have at least one primary key, so we define the id property as being **the primary key** of the table "User" thanks to the annotation [@PrimaryKey](https://developer.android.com/reference/android/arch/persistence/room/PrimaryKey.html) . Fast and efficient, right? :)

**Excerpt from Item.kt:**

@Entity(foreignKeys = arrayOf(ForeignKey(entity = User::class,

parentColumns = arrayOf("id"),

childColumns = arrayOf("userId"))),

indices = arrayOf(Index("userId")))

class Item (

@PrimaryKey(autoGenerate = true)

var id: Long = 0,

...

}

**Explanations:** The table "Item" is a little different, because we have added, inside the annotation @Entity, **the foreign-key/key-primary relationship** using the annotation [foreignKey](https://developer.android.com/reference/android/arch/persistence/room/ForeignKey.html) .

We’ve also defined a primary index to speed querying of this table using the [indices](https://developer.android.com/reference/android/arch/persistence/room/Indices.html) annotation.

As you may have noticed, we’ve added the parameter [autoGenerate](https://developer.android.com/reference/android/arch/persistence/room/PrimaryKey.html%23autoGenerate()) = true to the annotation @PrimaryKey: This will allow Room to **automatically generate a unique identifier** for each item saved... ;)

If you want to learn more about **how tables work** with Room, feel free to consult Android’s [official documentation](https://developer.android.com/training/data-storage/room/defining-data.html).

Nicely done! Now I’ll give you just the content of the **Adapter** and the **ViewHolder** that we will use subsequently through the **RecyclerView** of the activity **TodoListActivity**. As you can imagine, the RecyclerView will contain a list... of **Item**s! :D

Feel free to reread [the course dedicated to RecyclerView](https://openclassrooms.com/courses/recuperez-et-affichez-des-donnees-distantes/implementer-votre-premiere-recyclerview) if you want to refresh your memory about the **Adapter** and **ViewPager**.

Put these classes in the package **todolist**/.

**Class todolist/ItemViewHolder.kt:**

class ItemViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView), View.OnClickListener {

var textView: TextView

var imageView: ImageView

var imageButton: ImageButton

init {

textView = itemView.findViewById<View>(R.id.activity\_todo\_list\_item\_text) as TextView

imageView = itemView.findViewById<View>(R.id.activity\_todo\_list\_item\_image) as ImageView

imageButton = itemView.findViewById<View>(R.id.activity\_todo\_list\_item\_remove) as ImageButton

}

// FOR DATA

private var callbackWeakRef: WeakReference<ItemAdapter.Listener>? = null

fun updateWithItem(item: Item, callback: ItemAdapter.Listener) {

callbackWeakRef = WeakReference(callback)

textView.text = item.text

imageButton.setOnClickListener(this)

when (item.category) {

// TO VISIT

0 -> imageView.setBackgroundResource(R.drawable.ic\_room\_black\_24px)

// IDEAS

1 -> imageView.setBackgroundResource(R.drawable.ic\_lightbulb\_outline\_black\_24px)

// RESTAURANTS

2 -> imageView.setBackgroundResource(R.drawable.ic\_local\_cafe\_black\_24px)

}

if (item.selected) {

textView.paintFlags = textView.paintFlags or Paint.STRIKE\_THRU\_TEXT\_FLAG

} else {

textView.paintFlags = textView.paintFlags and Paint.STRIKE\_THRU\_TEXT\_FLAG.inv()

}

}

override fun onClick(view: View) {

val callback = callbackWeakRef!!.get()

callback?.onClickDeleteButton(adapterPosition)

}

}

**Explanations:** This class represents each line in the RecyclerView. I won’t explain it any more than that, because you should understand the essential of the Code without any trouble.

**Class todolist/ItemAdapter.kt:**

class ItemAdapter(private val callback: Listener) : RecyclerView.Adapter<ItemViewHolder>() {

// FOR DATA

private lateinit var items: List<Item>

// CALLBACK

interface Listener {

fun onClickDeleteButton(position: Int)

}

init {

this.items = ArrayList()

}

override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ItemViewHolder {

val context = parent.context

val inflater = LayoutInflater.from(context)

val view = inflater.inflate(R.layout.activity\_todo\_list\_item, parent, false)

return ItemViewHolder(view)

}

override fun onBindViewHolder(viewHolder: ItemViewHolder, position: Int) {

viewHolder.updateWithItem(this.items[position], this.callback)

}

override fun getItemCount(): Int {

return this.items.size

}

fun getItem(position: Int): Item {

return this.items[position]

}

fun updateData(items: List<Item>) {

this.items = items

this.notifyDataSetChanged()

}

}

**Explanations:** This class represents the Adapter making the link between the RecyclerView and the ViewHolder. Again, I won't explain it in any more detail, because this too is quite understandable on its own. If you have difficulties figuring out how the callback and its interface work, feel free to reread [the chapter of this course](https://openclassrooms.com/courses/recuperez-et-affichez-des-donnees-distantes/interagir-avec-la-recyclerview%23/id/r-4912758).

Now that everything seems ready, we will see in the next chapter how to **create our first SQL queries** on our SQLite database... ;)

### **Manipulate your data using the DAO**

Now that our entities are defined, it would be nice to be able to **manipulate** them through our SQLite database. For example, we would like to be able to **add** a thing to do into our database, **update** it or **delete** it... in short, do any of the [CRUD](https://fr.wikipedia.org/wiki/CRUD) actions!

Well, that’s just what we’ll do right away thanks to a [design pattern](https://fr.wikipedia.org/wiki/Patron_de_conception) called [Data Access Object](https://fr.wikipedia.org/wiki/Objet_d'accès_aux_données) (DAO).

Hold up, what's a design pattern? Why are you talking about DAO all of a sudden? :'(

Good question! A Design Pattern is a **template** or set of software **best practices** intended to help developers to design **the best solution** to a common software problem. No need to reinvent the wheel!

In our case, we have a problem: What is **the best way** to access data located in our SQLite database? In other words, the**"cleanest"** approach in object-oriented programming?



That’s where DAO comes in! :D This pattern allows us to **group together access to persistent data** in **separate classes**, rather than dispersing them. What if we applied this to our two tables User and Item ?

To do this, create a package called **database/** then a sub-package **DAO/,** and put the following two interfaces (note, not classes... ;)) into it: ItemDao and userDao .

**Interface database/dao/ItemDao.kt:**

@Dao

interface ItemDao {

@Query("SELECT \* FROM Item WHERE userId = :userId")

fun getItems(userId: Long): LiveData<List<Item>>

@Query("SELECT \* FROM Item WHERE userId = :userId")

fun getItemsWithCursor(userId: Long): Cursor

@Insert

fun insertItem(item: Item): Long

@Update

fun updateItem(item: Item): Int

@Query("DELETE FROM Item WHERE id = :itemId")

fun deleteItem(itemId: Long): Int

}

**Explanations:** Room requires us to use a particular format to create a DAO, asking us to create an interface for each DAO. Here, we have created the interface ItemDao whose **objective** and **responsibility** will be to consolidate all the CRUD actions for the item table .

In order to indicate the interface as a DAO class, we have added the annotation [@Dao](https://developer.android.com/reference/android/arch/persistence/room/Dao.html) at the top of it. Then we created the four CRUD actions via the following methods:

* getItems() : Used to retrieve the list of things to do (**Item**) for a user. We've used the annotation [@Query](https://developer.android.com/reference/android/arch/persistence/room/Query.html) to define the method as a SQL query. We return a **list of Items** of the type [LiveData](https://developer.android.com/reference/android/arch/lifecycle/LiveData.html), which I will explain in detail in the next part of this course... :)
* insertItem() : Used to create a new thing to do (**Item**) thanks to the annotation[@Insert](https://developer.android.com/reference/android/arch/persistence/room/Insert.html). Note that we passed an **Item** object directly as a parameter of the method. This object will not need to have a defined identifier, because remember from the previous chapter, Room **will generate** one for us!!!
* updateItem() : Used to update an existing thing to do (**Item**) using the annotation [@Update](https://developer.android.com/reference/android/arch/persistence/room/Update.html). Also note that we passed an **Item** object to it directly as a parameter. That object absolutely must have a defined identifier, so that Room can find in the DB and update it... ;)
* deleteItem() : Used to delete an existing thing to do (**Item**) in the DB. Note here that we have re-used the annotation @Query because we need to create a slightly more advanced SQL query. You can also go directly to an Item object and use the annotation [@Delete](https://developer.android.com/reference/android/arch/persistence/room/Delete.html) to delete it.

**Interface database/dao/UserDao.kt:**

@Dao

interface UserDao {

@Insert(onConflict = OnConflictStrategy.REPLACE)

fun createUser(user: User)

@Query("SELECT \* FROM User WHERE id = :userId")

fun getUser(userId: Long): LiveData<User>

}

**Explanations:** Here we have defined the DAO interface dedicated to manipulating the table User . We've added just two methods: One to create a new user ( createUser ) and one to retrieve a user ( getUser ). You’ll notice that we have added to the annotation @Insert , the parameter onConflict = OnConflictStrategy.REPLACE to **overwrite** an existing user with the same ID as the one you want to insert.

And that's all! :D Lastly, yes, we still need to configure a very important class whose role will be to **bind** all the classes/interfaces that we have previously created together, and especially to **configure** our database!

So I’ll leave it to you to create the class **SaveMyTripDatabase** in the package **database/**.

**Interface database/SaveMyTripDatabase.kt:\**

Class database/SaveMyTripDatabase.kt:

@Database(entities = arrayOf(Item::class, User::class), version = 1, exportSchema = false)

abstract class SaveMyTripDatabase : RoomDatabase() {

// --- DAO ---

abstract fun itemDao(): ItemDao

abstract fun userDao(): UserDao

companion object {

// --- SINGLETON ---

@Volatile

private var INSTANCE: SaveMyTripDatabase? = null

// --- INSTANCE ---

fun getInstance(context: Context): SaveMyTripDatabase? {

if (INSTANCE == null) {

synchronized(SaveMyTripDatabase::class.java) {

if (INSTANCE == null) {

INSTANCE = Room.databaseBuilder(context.applicationContext,

SaveMyTripDatabase::class.java, "MyDatabase.db")

.addCallback(prepopulateDatabase())

.build()

}

}

}

return INSTANCE

}

// ---

private fun prepopulateDatabase(): RoomDatabase.Callback {

return object : RoomDatabase.Callback() {

override fun onCreate(db: SupportSQLiteDatabase) {

super.onCreate(db)

val contentValues = ContentValues()

contentValues.put("id", 1)

contentValues.put("username", "Philippe")

contentValues.put("urlPicture", "https://oc-user.imgix.net/users/avatars/15175844164713\_frame\_523.jpg?auto=compress,format&q=80&h=100&dpr=2")

db.insert("User", OnConflictStrategy.IGNORE, contentValues)

}

}

}

}

}

**Explanations:** Here we've created an abstract class, inheriting from [RoomDatabase](https://developer.android.com/reference/android/arch/persistence/room/RoomDatabase.html) and defined by the annotation [@Database](https://developer.android.com/reference/android/arch/persistence/room/Database.html) . That same annotation will list the various tables (called "**entities**").

Inside this class, we have declared our two **DAO interfaces**. Next, we created a method, getInstance()inside a companion object to create a **singleton** of our class SaveMyTripDatabase.

Sooo... what’s a **Singleton**? :o

Well, it's another design pattern... ;) In this case, despite all our efforts, we've encountered a software problem: How to **JUST ONCE create** the class responsible for our database and get only a **single** reference instance?

Ah. But why do we need a **single instance** of our class managing our database?

Simply put, because if you have multiple instances of your database, you’ll potentially be allowing several instances to **manipulate the single file** of your database **at the same time**! If two instances open the SQLite file, and they **are trying to edit it at the same time**, it might lead to a slight problem, no? ;)

That’s why we create a [Singleton](https://en.wikipedia.org/wiki/Singleton_pattern)! It will create a new RoomDatabase object using its builder [Room.databaseBuilder](https://developer.android.com/reference/android/arch/persistence/room/Room.html%23databaseBuilder(android.content.Context,%20java.lang.Class) and will create a file that will contain our SQLite database. If that method is ever called back later, we will return only the reference of our database.

Thus, as you can see, we have added the [addCallback](https://developer.android.com/reference/android/arch/persistence/room/RoomDatabase.Builder.html%23addCallback(android.arch.persistence.room.RoomDatabase.Callback)) method to our Builder, which will enable us to **fill** it with a test user thanks to the method that we have created just below, prepopulateDatabase .

And there you go! Our SQLite database is configured correctly thanks to Room. Now all that’s left is for us to try it out... And that’s just what we’re going to do in the next chapter! ;)

### 

### **Test your database**

Before implementing the SQLite database in our activity **TodoListActivity**, we will **test** the various **CRUD calls** and verify that they're working correctly. This will also allow us to validate our DAO interfaces with various tests... :)

#### **Create a test class**

In order to begin testing our database's tables, we will focus on the class **ItemDAO** and validate most of its CRUD methods. For this, we will create a class called ItemDaoTest in the folder dedicated to our **instrumented tests**, **androidTest/**, so that they are run from a Android device (rather than on the JVM).

If the concept of an "**instrumented test**" is foreign to you, feel free to visit [the course](https://openclassrooms.com/courses/developpez-des-applications-robustes-et-fiables) that Florian has created about this subject.

But first, we must install a small library from Android, which will make it easier for us to implement our tests:

**Excerpt of build.gradle:**

dependencies {

...

// TESTING

androidTestImplementation "android.arch.core:core-testing:1.1.1"

}

Perfect! Now let’s move on to creating a class **responsible** for the tests carried out on ItemDAO.

**Class ItemDaoTest.kt:**

@RunWith(AndroidJUnit4::class)

class ItemDaoTest {

// FOR DATA

private lateinit var database: SaveMyTripDatabase = null

@Rule @JvmField

var instantTaskExecutorRule = InstantTaskExecutorRule()

@Before

@Throws(Exception::class)

fun initDb() {

this.database = Room.inMemoryDatabaseBuilder(InstrumentationRegistry.getContext(),

SaveMyTripDatabase::class.java)

.allowMainThreadQueries()

.build()

}

@After

@Throws(Exception::class)

fun closeDb() {

database.close()

}

}

**Explanations:** This test class will therefore be instrumented and run using [AndroidJUnitRunner](https://developer.android.com/training/testing/junit-runner.html) via the annotation @RunWith([AndroidJUnit4](https://developer.android.com/reference/android/support/test/runner/AndroidJUnit4.html).class). This test runner will load the package containing all of our tests on one Android device to run them.

Next we defined a rule using the annotation [@Rule](https://github.com/junit-team/junit4/wiki/rules) . As a reminder, a rule allows us to define **the manner** in which the tests will be executed. In our case here, we have used the [InstantTaskExecutorRule](https://developer.android.com/reference/android/arch/core/executor/testing/InstantTaskExecutorRule.html) to force the execution of each test **synchronously** (i.e. without moving them to a thread in the background).

Next, we created a method initDb() which will load to create an instance of our database, then place it in the variable database declared at the top of our class. This method will be called before each test is run, using the annotation [@Before](http://junit.sourceforge.net/javadoc/org/junit/Before.html) .

OK, but that's weird, the builder to generate our class Room is strange... o_O

Ah! You noticed... :) Yes, to facilitate the unit tests, Room provides us with a builder called [inMemoryDatabaseBuilder](https://developer.android.com/reference/android/arch/persistence/room/Room.html%23inMemoryDatabaseBuilder(android.content.Context,%20java.lang.Class). This builder **creates an instance** of our database directly **in memory** (rather than in a file on a device!). Isn't that handy?

#### **Testing our CRUD calls**

Now let’s move on to the tests. First, I’ll let you create (in your test package) a **utility object** that will enable us to more easily run methods that return values of the type [LiveData](https://developer.android.com/topic/libraries/architecture/livedata.html).

**Objectd LiveDataTestUtil.kt:**

object LiveDataTestUtil {

@Throws(InterruptedException::class)

fun <T> getValue(liveData: LiveData<T>): T {

val data = arrayOfNulls<Any>(1)

val latch = CountDownLatch(1)

val observer = object : Observer<T> {

override fun onChanged(o: T?) {

data[0] = o

latch.countDown()

liveData.removeObserver(this)

}

}

liveData.observeForever(observer)

latch.await(2, TimeUnit.SECONDS)

return data[0] as T

}

}

**Explanations:** This class is [provided by Google](https://github.com/googlesamples/android-architecture-components/blob/1532c562bb4d25f8469471478195160fd96b5db7/BasicSample/app/src/androidTest/java/com/example/android/persistence/LiveDataTestUtil.java) to help you more easily create tests involving the type LiveData, and especially to block the test’s execution until the result has been returned.

First let’s test the adding and retrieval of a new user in our SQLite database.

**Excerpt from ItemDaoTest:**

@RunWith(AndroidJUnit4.class)  
class ItemDaoTest {

...

// DATA SET FOR TEST

private val USER\_ID: Long = 1

private val USER\_DEMO = User(USER\_ID, "Philippe", "https://www.google.fr, ")

private val NEW\_ITEM\_PLACE\_TO\_VISIT = Item("Visit this dream place !", 0, USER\_ID)

private val NEW\_ITEM\_IDEA = Item("We could do dog sledding ?", 1, USER\_ID)

private val NEW\_ITEM\_RESTAURANTS = Item("This restaurant looks nice", 2, USER\_ID)

@Test

@Throws(InterruptedException::class)

fun insertAndGetUser() {

// BEFORE : Adding a new user

this.database.userDao().createUser(USER\_DEMO)

// TEST

val user = LiveDataTestUtil.getValue(this.database.userDao().getUser(USER\_ID))

assertTrue(user.username == USER\_DEMO.username && user.id == USER\_ID)

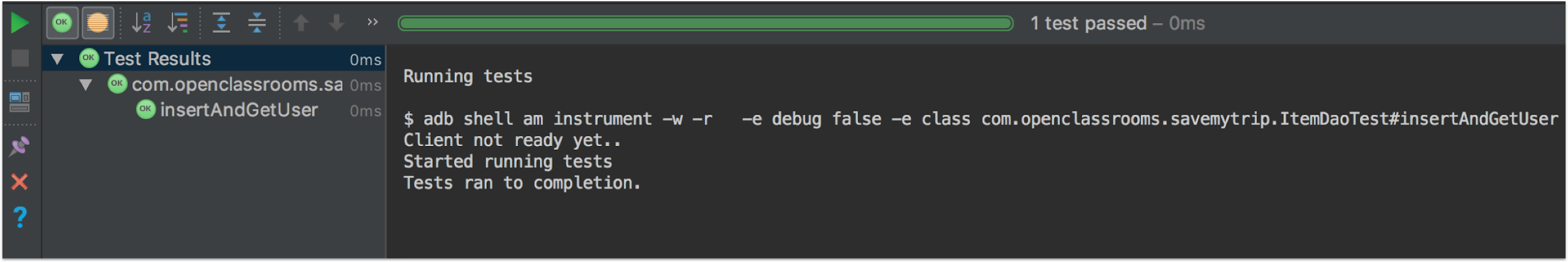
}

**Explanations:** Before creating a test, we declare and instantiate a data set that we will be likely to reuse in our various tests. Here we're simply creating a demo user... :)

Next, we’ll create a first test with the annotation [@Test](http://junit.sourceforge.net/javadoc/org/junit/Test.html). This test will first **insert** a new user into our database (with our DAO’s method createUser), then **retrieve** it, this time from the same database using the getUser method.

Once our User object has been recovered, we can **test** using the method assertTrue whether it matches the one we saved earlier. It's as simple as that! :D

Now let’s run the test! Congratulations, you’ve just created **your first test** on a database!

Running the test

However, for now, we’ve only tested the adding and retrieval of data from the table **User**, but not yet from the table **Item...** But because I’m so nice, I’ll give you the final tests so that you can test all of the CRUD methods of the **Item** table.

**Excerpt from ItemDaoTest:**

class ItemDaoTest {

...

private val NEW\_ITEM\_PLACE\_TO\_VISIT = Item("Visit this dream place !", 0, USER\_ID)

private val NEW\_ITEM\_IDEA = Item("We could do dog sledding ?", 1, USER\_ID)

private val NEW\_ITEM\_RESTAURANTS = Item("This restaurant looks nice", 2, USER\_ID)

...

@Test

@Throws(InterruptedException::class)

fun getItemsWhenNoItemInserted() {

// TEST

val items = LiveDataTestUtil.getValue(this.database!!.itemDao().getItems(USER\_ID))

assertTrue(items.isEmpty())

}

@Test

@Throws(InterruptedException::class)

fun insertAndGetItems() {

// BEFORE : Adding demo user & demo items

this.database.userDao().createUser(USER\_DEMO)

this.database.itemDao().insertItem(NEW\_ITEM\_PLACE\_TO\_VISIT)

this.database.itemDao().insertItem(NEW\_ITEM\_IDEA)

this.database.itemDao().insertItem(NEW\_ITEM\_RESTAURANTS)

// TEST

val items = LiveDataTestUtil.getValue(this.database.itemDao().getItems(USER\_ID))

assertTrue(items.size == 3)

}

@Test

@Throws(InterruptedException::class)

fun insertAndUpdateItem() {

// BEFORE : Adding demo user & demo items. Next, update item added & re-save it

this.database.userDao().createUser(USER\_DEMO)

this.database.itemDao().insertItem(NEW\_ITEM\_PLACE\_TO\_VISIT)

val itemAdded = LiveDataTestUtil.getValue(this.database.itemDao().getItems(USER\_ID))[0]

itemAdded.selected = true

this.database.itemDao().updateItem(itemAdded)

//TEST

val items = LiveDataTestUtil.getValue(this.database.itemDao().getItems(USER\_ID))

assertTrue(items.size == 1 && items[0].selected)

}

@Test

@Throws(InterruptedException::class)

fun insertAndDeleteItem() {

// BEFORE : Adding demo user & demo item. Next, get the item added & delete it.

this.database.userDao().createUser(USER\_DEMO)

this.database.itemDao().insertItem(NEW\_ITEM\_PLACE\_TO\_VISIT)

val itemAdded = LiveDataTestUtil.getValue(this.database.itemDao().getItems(USER\_ID))[0]

this.database.itemDao().deleteItem(itemAdded.id)

//TEST

val items = LiveDataTestUtil.getValue(this.database.itemDao().getItems(USER\_ID))

assertTrue(items.isEmpty())

}

}

**Explanations:** These tests are meant to be quite easy to read and understand... :) So I won’t explain them in detail. Just know that we're using all the same methods from our ItemDao interface and we’re checking to see if they return the information that they should.

Beginning in the next part of this course, we will see how to integrate the calls to our database directly from our **TodoListActivity** controller in as clean a manner as possible, using **Architecture Components**... ;)

Since this chapter is shorter, take the time to **read** and **reread** your code, in order to understand the logic behind it. And don't forget the golden rule of any good developer, **Practice Makes Perfect**! Good luck... :ange:

## **Hone your app’s architecture**

### **Discover Architecture Components**

If you’re reading this course, it’s because you're far enough along in developing mobile apps for Android... :)

As you go about developing mini-apps through the courses in our [Android curriculum](https://openclassrooms.com/paths/developpeur-se-d-applications-android), you might find yourself asking the question: ***Is there a software architecture that is optimal for developping a Android app?***

There is! Because it’s true that sometimes, my controllers (activities and/or fragments) seem to me to be a bit too big... :'(

Exactly! This phenomenon, known in iOS by the name "**Massive View Controller**", occurs when one of your controllers (activities and/or fragments) contains **too much code** and becomes almost **unreadable** or even **unstable**, and virtually **untestable**.

It’s true that I tend to write a lot of things in my activities... But it must be said that the smallest feature in an Android app takes up many lines of code!

We agree! :D In addition, as you have probably already noticed on Android, your controllers have a fairly unique **lifecycle**. That’s because these controllers can be **destroyed at any time**, taking with them all of the data that they contained. This may occur during **rotations** or when a **telephone call** arrives, or simply when the phone has no more **battery**... And these cases are very complicated for us developers to handle.

But what do the police do? (Or Google for that matter!) >_<

Early on, not much... and therein lay the problem! Android developers were left to their own devices, and tried more or less to apply a certain architecture onto their project, following development best practices like “[**separation of concerns (SoC)**](https://en.wikipedia.org/wiki/Separation_of_concerns)" as best they could: **one class** = **one responsibility, one method = one particular action,** etc...

Some have even decided to dispense with the MVC architecture and opt for “cleaner” architectures like [MVP](https://en.wikipedia.org/wiki/Model–view–presenter), [MVVM](https://en.wikipedia.org/wiki/Model–view–viewmodel), or [Clean Architecture](https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html), which vary in how widely they are adopted by the Android community... You’ll find [examples at this link](https://github.com/googlesamples/android-architecture).

It was in this tense environment that the teams at Google decided to release a **set of libraries** in late 2017 to more easily create apps that were **robust, testable, and maintainable** over time... which they called: The[**Android Architecture Components**](https://openclassrooms.com/courses/4568746/parts/5106569/)**!**

##### **What are the two founding principles of the Architecture Components?**

In order to create the best possible architecture for an Android app, Architecture Components is based on two **founding principles**:

1. Controllers (Activities or Fragments) should be used **ONLY** to **manipulate the graphical user interface** (creation, updating, etc...) and **interact** **with the operating system** (launching a new activity, for example). That’s it! The rest of the business code must absolutely be pushed off into separate classes.
2. The graphical user interface should generally be **updated** **from a class template (**[**POJO**](https://fr.wikipedia.org/wiki/Plain_old_Java_object), preferably **persistable**, and **reflect it (modeling)** as faithfully as possible. The goal is to **cope** more easily with a loss of connection to the Internet or the destruction of your application, by recovering the persisted model **directly from the storage of your phone** (from a database, for example... ;)).

Try to always keep these **two principles in mind** when you're developing a Android application. This will allow you to **naturally** ask the right questions in terms of the architecture.

##### **What elements are Architecture Components made up of?**

****Example Android Architecture Components

In order to **streamline** the controllers (activities and fragments) of our Android applications as much as possible, we had to move certain actions into dedicated classes. Thus, Android’s teams have chosen to offer the following architecture:

1. **The controller:** Your controller (activity or fragment) will be dedicated to **manipulating** your GUI, nothing more nor less.
2. [**The ViewModel**](https://developer.android.com/topic/libraries/architecture/viewmodel.html)**:** The controller will however implement a class of the type **ViewModel**, whose role will be to **provide the controller with data** used by the GUI. One of the unique features of the class ViewModel is its ability to **"survive" changes in configuration** such as the rotation of the screen without losing its data... :magicien:
3. [**LiveData**](https://developer.android.com/topic/libraries/architecture/livedata.html)**:** The data present within the ViewModel will be primarily **LiveData** which is an observable data holder class that allows us to **more easily "observe" their changes** while **respecting the life cycle** of our application’s activities and fragments. The type LiveData corresponds in some ways to the notion of Observable in RxJava, which we studied together [in a previous course](https://openclassrooms.com/courses/recuperez-et-affichez-des-donnees-distantes/perfectionner-votre-code-grace-la-programmation-reactive-et-rxjava), except that we ourselves do not need to manage the process of unsubscribing, and that LiveData has been designed for Android and its life cycle. If on the other hand, you want to continue using RxJava rather than LiveData, know that you absolutely can! :)
4. **The Repository:** Inside each ViewModel, we will find one or more classes of the type **Repository,** a [fairly common](https://msdn.microsoft.com/en-us/library/ff649690.aspx) design pattern**.** Its role is a bit particular because it we will serve as a **mediator** between the ViewModel and the various data sources.
5. **Data sources:** Inside each repository, we will have the different classes used to access our data, more commonly known as [Data Sources](https://en.wikipedia.org/wiki/Data_source). For example, these classes can take the form of **DAOs**, such as those that we created in the previous chapters, or even **communication interfaces** with APIs, as we have been able to do [in this course](https://openclassrooms.com/courses/recuperez-et-affichez-des-donnees-distantes/ameliorer-limplementation-de-vos-requetes-reseaux).

I know, all this is a bit theoretical! Moreover, it’s very likely that you’re wondering why there’s **such a large cut level**... In fact, you should be aware that the **more your code is cut in pieces**, the more **scalable**, **testable** and **legible** it will be.

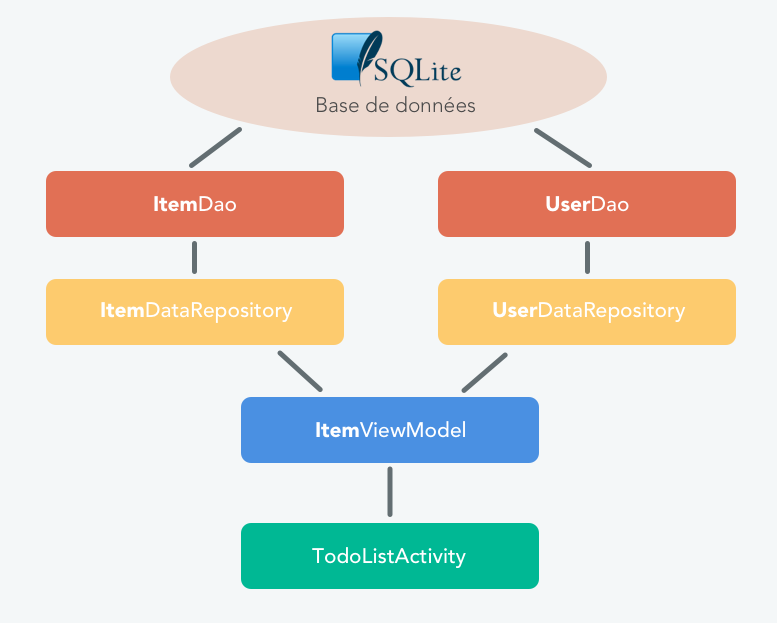
In mini-apps, it’s true that the result might not be obvious and it will be difficult to realize. However, when your project begins to grow and take up **hundreds of thousands of lines of code**, you’ll be very pleased to have a clean, maintainable architecture, believe you me... ;)

So, let’s get serious and implement all this in the next chapter, through our mini-application SaveMyTrip!

### **Define your app’s architecture using a ViewModel**

Now that we know a little more about [**Android Architecture Components**](https://developer.android.com/topic/libraries/architecture/index.html), we’re going to be able to implement some of its elements in our mini-app SaveMyTrip.

The objective is therefore **to apply** Architecture Components to our application, creating the classes**Repository** and **ViewModel** and then calling them inside our activity **TodoListActivity**.

Desired architecture

In the previous part of this course we developed the two classes that we will serve as **data sources**, namely the classes **ItemDao** and **UserDao**.

#### **Java 8 support**

First, we’re going to ensure our code is compatible with Java 8 so that we can use **Lambdas**. Feel free to read [this chapter of the Java course](https://openclassrooms.com/courses/apprenez-a-programmer-en-java/classes-anonymes-interfaces-fonctionnelles-lambdas-et-references-de-methode%23/id/r-5012615) which covers that subject. To summarize, Lambdas will allow us to write **less code**, in a much **more readable** way... :)

// WITHOUT LAMBDAS

button.setOnClickListener(object: View.OnClickListener {

override fun onClick(v: View?) {

Toast.makeText(parent, "Button clicked", Toast.LENGTH\_LONG).show();

}

});

//WITH LAMBDAS

button.setOnClickListener { Toast.makeText(parent, "Button clicked", Toast.LENGTH\_LONG).show() }

Now resume your application and edit your **build.gradle** file as follows...

**Excerpt of build.gradle:**

android {

...

compileOptions {

targetCompatibility 1.8

sourceCompatibility 1.8

}

}

There you have it, your app now supports Java 8 and Lambdas... ;)

#### **Creating Repositories**

You may have noticed that our **data source** is currently the **SQLite database**, manipulated via the different DAOs that we created previously. Going by the same logic as for Architecture Components, we will now create two classes, **ItemDataRepository** and **UserDataRepository**, which we’ll then use in a ViewModel, of course... :)

To do this, create a new package and call it **repositories/**, then put the following files into it:

**Class repositories/UserDataRepository.kt:**

class UserDataRepository(private val userDao: UserDao) {

// --- GET USER ---

fun getUser(userId: Long): LiveData<User> {

return this.userDao.getUser(userId)

}

}

**Class repositories/ItemDataRepository.kt:**

class ItemDataRepository(private val itemDao: ItemDao) {

// --- GET ---

fun getItems(userId: Long): LiveData<List<Item>> {

return this.itemDao.getItems(userId)

}

// --- CREATE ---

fun createItem(item: Item) {

itemDao.insertItem(item)

}

// --- DELETE ---

fun deleteItem(itemId: Long) {

itemDao.deleteItem(itemId)

}

// --- UPDATE ---

fun updateItem(item: Item) {

itemDao.updateItem(item)

}

}

**Explanations:** These two classes are simple enough on their own, since they retrieve, from their constructor, a DAO that they reuse in their public methods... :) The purpose of the Repository is really to **isolate** the data source (DAO) from the ViewModel, so that the model **does not directly manipulate** the data source.

#### **Creating the ViewModel**

Let’s keep implementing Architecture Components within our application by creating the class **ItemViewModel**,which we will place in the package **todolist/** and which will subsequently be integrated into our activity **TodoListActivity**.

**Class todolist/ItemViewModel.kt:**

class ItemViewModel(private val itemDataSource: ItemDataRepository,

private val userDataSource: UserDataRepository,

private val executor: Executor) : ViewModel() {

// DATA

private var currentUser: LiveData<User>? = null

fun init(userId: Long) {

if (currentUser != null) {

return

}

currentUser = userDataSource.getUser(userId)

}

// -------------

// FOR USER

// -------------

fun getUser(userId: Long): LiveData<User>? {

return currentUser

}

// -------------

// FOR ITEM

// -------------

fun getItems(userId: Long): LiveData<List<Item>> {

return itemDataSource.getItems(userId)

}

fun createItem(item: Item) {

executor.execute { itemDataSource.createItem(item) }

}

fun deleteItem(itemId: Long) {

executor.execute { itemDataSource.deleteItem(itemId) }

}

fun updateItem(item: Item) {

executor.execute { itemDataSource.updateItem(item) }

}

}

**Explanations:** So that’s what our ViewModel looks like! :D First of all, it inherits from the class [ViewModel](https://developer.android.com/topic/libraries/architecture/viewmodel.html). Next, we declare in it, as class variables, our two previously created Repositories as well as a variable of the type [Executor](https://developer.android.com/reference/kotlin/java/util/concurrent/Executor.html), which will help us **run certain methods in the background**. These three variables are instantiated directly from the **constructor** of the class.

We've also created an init() method, in order to initialize our ViewModel as soon as the activity is created and which will therefore be called inside of its method onCreate() .

OK, but why do you check in this method whether the user already exists in the ViewModel?

Well, simply because the ViewModel **stores its data in "memory"**, **even if the activity that called it is destroyed**, such as after a rotation... :) That’s the whole reason for the ViewModel! So, after a rotation of the activity TodoListActivity, we will not **need to re-retrieve** the user from the database **if** the user was **previously** stored in the ViewModel.

Next, we have created different methods to perform actions on our database (representing our data source). We use the class **Executor** in order to **asynchronously** perform update requests for our SQLite tables.

It is important to recall that **all the queries** made on the SQLite database using Room are run [**synchronously**](https://openclassrooms.com/courses/recuperez-et-affichez-des-donnees-distantes/decouvrir-les-taches-asynchrones%23/id/r-4913802)! This can be problematic if you're retrieving large volumes of data... Hence the importance of our putting the methods into separate threads.

Furthermore, this is also the reason why we use the type **LiveData** in the methods getItems and getUser of our **DAOs**, in order to **automatically** benefit from **asynchronous retrieval**... ;)

#### **Editing the activity**

Before declaring our ViewModel in our activity **TodoListActivity**, we need to build it... Because you may have already noticed that the **constructor** of the class **ItemViewModel** requested as a parameter:

* The class **ItemDataRepository** (itself requesting the class **ItemDao** as a parameter)
* The class **UserDataRepository** (itself requesting the class **UserDao** as a parameter)
* The class **Executor**.

This may be a lot of things to declare and instantiate in our activity... and as a reminder, it’s not supposed to deal with such things! We want to be as **streamlined** as possible... >_<

Which is fortunate, as we can move the construction of our ViewModel into a **Factory** class!

And just what is that? Another design pattern, I’m guessing?

Yes indeed... :) But don’t panic! A [factory](https://fr.wikipedia.org/wiki/Fabrique_(patron_de_conception)) is simply a pattern used to **delegate the creation** of one class **to another**. So in our case, we're not going to directly create our ItemViewModel class in our activity; instead, we're going to assign that task to the class... ViewModelFactory , which we’ll put in a package that we’ll call **injections/**.

**Class injections/ViewModelFactory.kt:**

class ViewModelFactory(private val itemDataSource: ItemDataRepository,

private val userDataSource: UserDataRepository,

private val executor: Executor) : ViewModelProvider.Factory {

@Suppress("UNCHECKED\_CAST")

override fun <T : ViewModel> create(modelClass: Class<T>): T {

return if (modelClass.isAssignableFrom(ItemViewModel::class.java)) {

ItemViewModel(itemDataSource, userDataSource, executor) as T

} else {

throw IllegalArgumentException("Unknown ViewModel class")

}

}

}

**Explanations:** We've created a class ViewModelFactory , implementing the interface [ViewModelProvider.Factory](https://developer.android.com/reference/android/arch/lifecycle/ViewModelProvider.Factory.html) created by Google, which will subsequently be used to declare our ViewModel in our activity. Here we define a constructor containing the objects we need to correctly instantiate our class **ItemViewModel**.

Wait, I don't get it! We created the same constructor for the class **ViewModelFactory** as for the class **ItemViewModel**. What good is that?

For now, this allows us to **consolidate the process of creating** our ViewModels into a **dedicated** Factory, **ViewModelFactory**. This way, if at some later point we want to create another ViewModel, such as UserViewModel , we will declare it here, in the same Factory, inside the method create() ... :)

In order to better visualize the whole logic of this fairly complex **encapsulation process**, we will complete the implementation by creating a class that will be responsible for **injecting** each object in the constructor of our Factory: This process is often referred to as [**dependency injection**](https://en.wikipedia.org/wiki/Dependency_injection). I won't be talking about this **very advanced** concept right now, but be aware that there are specialty libraries like [Dagger2](https://github.com/google/dagger) that can do this in a more **automated** way.

So, let’s create the **Injection.java** object in the package **injection/**.

**Object injection/Injection.kt:**

object Injection {

private fun provideItemDataSource(context: Context): ItemDataRepository {

val database = SaveMyTripDatabase.getInstance(context)

return ItemDataRepository(database!!.itemDao())

}

private fun provideUserDataSource(context: Context): UserDataRepository {

val database = SaveMyTripDatabase.getInstance(context)

return UserDataRepository(database!!.userDao())

}

private fun provideExecutor(): Executor {

return Executors.newSingleThreadExecutor()

}

fun provideViewModelFactory(context: Context): ViewModelFactory {

val dataSourceItem = provideItemDataSource(context)

val dataSourceUser = provideUserDataSource(context)

val executor = provideExecutor()

return ViewModelFactory(dataSourceItem, dataSourceUser, executor)

}

}

**Explanations:** This class will be responsible for **providing** objects **already built**, in a centralized way. For instance, when we want to create objects present in this class anywhere in our application, we’ll directly invoke its public methods instead of making a new MyObject(). This allows you to make our code even **more modular**, by avoiding the creation of **strong dependencies** between each of our classes.

Don’t worry if you don’t understand all these very advanced notions yet! :ange: I wanted to show you **at least once** so you can see what this might look like.

Finally, let’s edit our **TodoListActivity** activity to add our ViewModel...

**Excerpt from TodoListActivity.kt:**

class TodoListActivity : BaseActivity(), ItemAdapter.Listener {

...

// 1 – Variables

// FOR UI

private lateinit var recyclerView: RecyclerView

private lateinit var spinner: Spinner

private lateinit var editText: EditText

private lateinit var profileImage: ImageView

private lateinit var profileText: TextView

private lateinit var addButton: Button

// FOR DATA

private lateinit var itemViewModel: ItemViewModel

private lateinit var adapter: ItemAdapter = null

private val USER\_ID = 1

...

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

editText = findViewById(R.id.todo\_list\_activity\_edit\_text)

profileImage = findViewById(R.id.todo\_list\_activity\_header\_profile\_image)

profileText = findViewById(R.id.todo\_list\_activity\_header\_profile\_text)

addButton = findViewById(R.id.todo\_list\_activity\_button\_add)

addButton.setOnClickListener{ createItem() }

val ab = supportActionBar

ab!!.setDisplayHomeAsUpEnabled(true)

configureToolbar()

configureSpinner()

// 1 - Configure RecyclerView & ViewModel

configureRecyclerView();

configureViewModel();

// 2 - Get current user & items from Database

getCurrentUser(USER\_ID);  
 getItems(USER\_ID);

}

// -------------------

// ACTIONS

// -------------------

@OnClick(R.id.todo\_list\_activity\_button\_add)

fun onClickAddButton() {

// 3 – Create item after user clicks on add button

createItem()

}

// 4 - Delete item after user clicks on delete button

override fun onClickDeleteButton(position: Int) {

deleteItem(this.adapter!!.getItem(position))

}

// -------------------

// DATA

// -------------------

// 5 - Configuring ViewModel

private fun configureViewModel() {

val mViewModelFactory = Injection.provideViewModelFactory(this)

itemViewModel = ViewModelProviders.of(this, mViewModelFactory).

get(ItemViewModel::class.java)

itemViewModel.init(USER\_ID.toLong())

}

// 7 - Get Current User  
 private fun getCurrentUser(userId: Int) {

itemViewModel.getUser(userId.toLong())!!.observe(this, Observer

{ this.updateHeader(it) })

}

// 7 - Get all items for a user

private fun getItems(userId: Int) {

itemViewModel.getItems(userId.toLong()).observe(this, Observer

{ this.updateItemsList(it) })

}

// 7 - Create a new item

private fun createItem() {

val item = Item(this.editText.text.toString(), this.spinner.selectedItemPosition,

USER\_ID.toLong(), false)

editText.setText("")

itemViewModel.createItem(item)

}

// 7 - Delete an item  
 private fun deleteItem(item: Item) {

itemViewModel.deleteItem(item.id)

}

// 7 - Update an item (selected or not)

private fun updateItem(item: Item) {

item.selected = !item.selected

itemViewModel.updateItem(item)

}

// -------------------

// UI

// -------------------

private fun configureSpinner() {

spinner = findViewById(R.id.todo\_list\_activity\_spinner)

val adapter = ArrayAdapter.createFromResource(this, R.array.category\_array,

android.R.layout.simple\_spinner\_item)

adapter.setDropDownViewResource(android.R.layout.simple\_spinner\_dropdown\_item)

spinner.adapter = adapter

}

// 8 - Configure RecyclerView

private fun configureRecyclerView() {

itemAdapter = ItemAdapter(this)

recyclerView = findViewById(R.id.todo\_list\_activity\_recycler\_view)

recyclerView.adapter = itemAdapter

recyclerView.layoutManager = LinearLayoutManager(this)

val support: ItemClickSupport = ItemClickSupport.addTo(recyclerView, R.layout.activity\_todo\_list\_item)

support.setOnItemClickListener(this)

}

override fun onItemClicked(recyclerView: RecyclerView, position: Int, v: View) {

updateItem(itemAdapter.getItem(position))

}

// 9 - Update header (username & picture)

private fun updateHeader(user: User?) {

profileText.text = user?.username

Glide.with(this).load(user?.urlPicture).

apply(RequestOptions.circleCropTransform()).into(profileImage)

}

// 10 - Update the list of items

private fun updateItemsList(items: List<Item>?) {

itemAdapter.updateData(items!!)

}

}

**Explanations:** In our activity TodoListActivity, we first declared **(1)** different variables for our user interface items, ViewModel ItemViewModel , our adapter, and a variable representing our user’s **identifier** (for testing purposes).

Then, in the activity onCreate(), we call methods to confgure our recycler view and view model **(1).** We also have calls to get the current user object and get the list of items that user has created. **(2)**

In the add button click listener, we added a call to call a createItem method **(3)**.

We also **implemented** the interface ItemAdapter.Listener in our activity to manage clicking on the Delete button **(4).**

Next, we created a method **(5)** called configureViewModel, which we will use to initialize our ViewModel. As you can see, we’re initializing a variable ViewModelFactory from our class Injection, which we created earlier. Using this Factory, we’ll be able to instantiate our variable ItemViewModel, without needing to go directly through its constructor... :) Finally, once our ViewModel has been retrieved, we call its method init() in order to first retrieve the user and store it in ViewModel.

We've also created different private methods **(7)** calling on the public methods of our ViewModel in order to **observe their result**. For **Get** methods, we have used the method observe() to be **alerted automatically** if the result in the database changes... :)

We've also used the lambdas to **reduce** our expression and call the method updateHeader() when a change occurs.

// WITH LAMBDAS

private fun getCurrentUser(userId: Int) {

itemViewModel.getUser(userId.toLong())!!.observe(this, Observer { this.updateHeader(it) })

}

// WITHOUT LAMBDAS  
private fun getCurrentUser(userId: Int) {

itemViewModel.getUser(userId.toLong())!!.observe(this,object: Observer<User> {

override fun onChanged(it: User?) {

updateHeader(it)

}

})

}

We also created **(8)** a method configureRecyclerView() enabling us, as its name indicates, to configure the RecyclerView that we use to display our list of to do items.

We've also added two methods **(9)** and **(10)** updating our graphical user interface, when we retrieve an object representing our user (User ) or a list of things to do ( List<item> ).

.

Run your app, and play a little with it. It should now be 100% functional... :D

Great! However, I was wondering if there was any way to view our SQLite database from our PC, in order to facilitate its debugging?

Of course! I recommend you take a look at the library [**Stetho**](https://facebook.github.io/stetho/) created by Facebook, which among other things will allow you to **view** the contents of your database from your web browser... :)

Take the time to **read** and **reread** this code in order to familiarize yourself with all the concepts it refers to. And don't forget the golden rule of any good developer, **Practice Makes Perfect**! Good luck... :ange:

### 

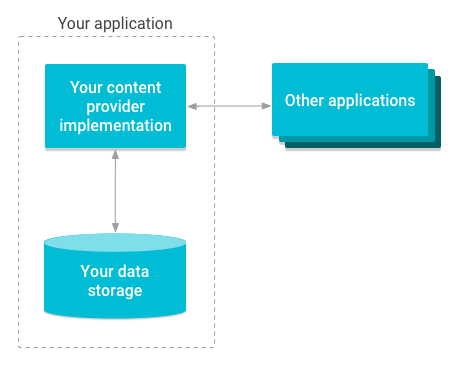
### **Expose your data with a ContentProvider**

Now that your app is functional, it would be interesting to be able to securely **expose** the contents of its SQLite database to other apps than your own.

Alright, but I don’t quite understand why we’d need to? :euh:

Well, sometimes an Android application may **publicly** and **securely** offer access to some of its data. For example, in our case, imagine that a company such as [Tripadvisor](https://www.tripadvisor.fr/) or [Booking.com](https://www.booking.com/) contacts you and offers you a **paid partnership** to view the list of things to do for each of your users, in order to offer them discounts and deals via their mobile apps: You would then need to provide a **secure**, **limited** access to your SQLite database! :)

This exposure is possible on Android via a [Content Provider](https://developer.android.com/guide/topics/providers/content-providers.html), allowing you to share content that you previously defined with applications other than your own.



Overview of how a Content Provider works

In a previous chapter, we saw how to publicly expose a file using the class [FileProvider](https://developer.android.com/reference/android/support/v4/content/FileProvider.html), inheriting from the class [ContentProvider](https://developer.android.com/reference/android/content/ContentProvider.html).

This time, in order to **expose** our SQLite database, we will directly use the class ContentProvider... :) Once it has been configured, outside apps will, for the purpose of accessing the exposed data, **need to provide it with a URI** that it **will analyze** to return the appropriate data.

Start by creating the package **provider/** in which you’ll create the class **ItemContentProvider**.

**Class provider/ItemContentProvider.kt:**

class ItemContentProvider : ContentProvider() {

override fun onCreate(): Boolean {

return true

}

override fun query(uri: Uri, projection: Array<String>?,

selection: String?,

selectionArgs: Array<String>?,

sortOrder: String?): Cursor? { return null;

}

override fun getType(uri: Uri): String? {

return null;

}

override fun insert(uri: Uri, contentValues: ContentValues?): Uri? {

return null;

}

override fun delete(uri: Uri, s: String?, strings: Array<String>?): Int {

return 0;

}

override fun update(uri: Uri,

contentValues: ContentValues?,

s: String?,

strings: Array<String>?): Int {

return 0;

}

}

**Explanations:** Here we’ve created a class, **ItemContentProvider,** inheriting from ContentProvider, whose purpose will be to expose the data of our SQLite database. A class inheriting from ContentProvider absolutely must implement the following six methods:

* [onCreate()](https://developer.android.com/reference/android/content/ContentProvider.html%23onCreate()) : Represents the Content Provider’s entry point. This way, you’ll be able to initialize different variables that will be useful to you later on.
* [query()](https://developer.android.com/reference/android/content/ContentProvider.html%23query(android.net.Uri,%20java.lang.String%5b%5d,%20java.lang.String,%20java.lang.String%5b%5d,%20java.lang.String)): This method will take a URI as input and retrieve the data (via a [Cursor](https://developer.android.com/reference/android/database/Cursor.html)) from the destination of your choice (in our case, our SQLite database).
* [getType()](https://developer.android.com/reference/android/content/ContentProvider.html%23getType(android.net.Uri)): This method allows you to return the type [MIME](https://developer.android.com/guide/topics/providers/content-provider-creating.html%23MIMETypes) associated with the URI to more accurately identify the type of data that will be returned.
* [insert()](https://developer.android.com/reference/android/content/ContentProvider.html%23insert(android.net.Uri,%20android.content.ContentValues)): This method will take a URI as input and **insert** data in [ContentValues](https://developer.android.com/reference/android/content/ContentValues.html) format into the destination of our choice (in our case, our SQLite database).
* [delete()](https://developer.android.com/reference/android/content/ContentProvider.html%23delete(android.net.Uri,%20java.lang.String,%20java.lang.String%5b%5d)): This method will take a URI as input and **delete** data in ContentValues format from the destination of our choice (in our case, our SQLite database).
* [update()](https://developer.android.com/reference/android/content/ContentProvider.html%23update(android.net.Uri,%20android.content.ContentValues,%20java.lang.String,%20java.lang.String%5b%5d)): This method will take a URI as input and **update** data in ContentValues format in the destination of our choice (in our case, our SQLite database).

For now, these methods are empty... but don’t worry, we’ll be filling them in good time! :) First, we need to declare our ContentProvider in the **manifest** of our application in order to activate it.

**Excerpt from AndroidManifest.xml:**

<?xml version="1.0" encoding="utf-8"?>

<manifest

xmlns:android="http://schemas.android.com/apk/res/android"

package="com.openclassrooms.savemytripkt">

...

<application

...

<provider

android:name=".provider.ItemContentProvider"

android:authorities="com.openclassrooms.savemytripkt.provider"

android:exported="true"/>

</application>

</manifest>

**Explanations:** On the same principle as what we did in the previous chapter concerning the FileProvider, we must declare the ContentProvider in the manifest of our application in order to make it **accessible** to other applications.

Now that our ContentProvider has been correctly declared, we will be able to **configure** it to give it access to our SQLite database, and notably our **Item** table . To do so, we will slightly edit the DAO managing our **Item** table in order to be able to return an object of the type [Cursor](https://developer.android.com/reference/android/database/Cursor.html), which can more easily be manipulated by the ContentProvider.

**Excerpt from ItemDao.java:**

@Dao  
interface ItemDao {

@Query("SELECT \* FROM Item WHERE userId = :userId")

fun getItemsWithCursor(userId: Long): Cursor

...

}

Next, we’ll edit our template class **Item** in order to create a public method, enabling it to turn a [ContentValues](https://developer.android.com/reference/android/content/ContentValues.html) object into an **Item** object.

It’s true! Outside apps **do not know** the exact structure of your Item class, so they send your ContentProvider a general object of the type **ContentValues**, leaving it up to you to turn it into the desired object (in our case, an **Item**).

**Excerpt from Item.kt:**

class Item {

...

companion object {

// --- UTILS ---

fun fromContentValues(values: ContentValues): Item {

val item = Item()

if (values.containsKey("text")) item.text = values.getAsString("text")

if (values.containsKey("category")) item.category = values.getAsInteger("category")

if (values.containsKey("isSelected")) item.selected = values.getAsBoolean("isSelected")

if (values.containsKey("userId")) item.userId = values.getAsLong("userId")

return item

}

}

}

**Explanations:** The class ContentValues is actually a fairly basic dictionary that allows us to retrieve a value from a key... :) We will then be placing this value into the corresponding property of the **Item** object.

Now let’s edit our ContentProvider in order to expose our **Item** table.

**Class ItemContentProvider.kt:**

class ItemContentProvider : ContentProvider() {

companion object {

// FOR DATA

val AUTHORITY = "com.openclassrooms.savemytrip.provider"

val TABLE\_NAME = Item::class.java.simpleName

val URI\_ITEM = Uri.parse("content://$AUTHORITY/$TABLE\_NAME")

}

override fun onCreate(): Boolean {

return true

}

override fun query(uri: Uri, projection: Array<String>?,

selection: String?,

selectionArgs: Array<String>?,

sortOrder: String?): Cursor? {

if (context != null) {

val userId = ContentUris.parseId(uri)

val db = SaveMyTripDatabase.getInstance(context)

val cursor = db!!.itemDao().getItemsWithCursor(userId)

cursor.setNotificationUri(context.contentResolver, uri)

return cursor

}

throw IllegalArgumentException("Failed to query row for uri $uri")

}

override fun getType(uri: Uri): String? {

return "vnd.android.cursor.item/$AUTHORITY.$TABLE\_NAME"

}

override fun insert(uri: Uri, contentValues: ContentValues?): Uri? {

if (context != null) {

val db = SaveMyTripDatabase.getInstance(context)

val id = db!!.itemDao().insertItem(Item.fromContentValues(contentValues!!))

if (id != 0L) {

context.contentResolver.notifyChange(uri, null)

return ContentUris.withAppendedId(uri, id)

}

}

throw IllegalArgumentException("Failed to insert row into $uri")

}

override fun delete(uri: Uri, s: String?, strings: Array<String>?): Int {

if (context != null) {

val db = SaveMyTripDatabase.getInstance(context)

val count = db!!.itemDao().deleteItem(ContentUris.parseId(uri))

context.contentResolver.notifyChange(uri, null)

return count

}

throw IllegalArgumentException("Failed to delete row into $uri")

}

override fun update(uri: Uri, contentValues: ContentValues?, s: String?, strings: Array<String>?): Int {

if (context != null) {

val db = SaveMyTripDatabase.getInstance(context)

val count = db!!.itemDao().updateItem(Item.fromContentValues(contentValues!!))

context.contentResolver.notifyChange(uri, null)

return count

}

throw IllegalArgumentException("Failed to update row into $uri")

}

}

**Explanations:** Here we have completed all the methods of our ContentProvider. First, we declared some value variables in a companion object that help us identify the authority defining our **ContentProvider**, the **name of the table** that we will query, and the **base URI** that will need to be learned to communicate with it.

In general, you'll notice that we follow a **certain logic** for each of the methods of our ContentProvider:

1. First, we conduct an operation (create, retrieve, update, and delete) on our SQLite database, and more particularly on the **Item** table, using our **SaveMyTripDatabase** object that will call the DAO **ItemDao**.
2. Second, we re-send or update **the URI** of the manipulated resource in order to inform the user that the operation was a success.

For the methods update and delete , we return the number of lines that were impacted by the operation in question.

At the end of each of our methods, we return an **exception** IllegalArgumentException if at any point the ContentProvider cannot fully complete the operation.

Alright! But still, I wonder how we’re going to test our ContentProvider... I’ll need to create a second app, right? :)

In fact, there is a simpler way to achieve this... particularly with instrumented tests!:D To do so, I created a test class for you, **ItemContentProviderTest**, which will enable us to easily check if our ContentProvider is working as it should.

**Class ItemContentProviderTest.kt:**

@RunWith(AndroidJUnit4::class)

class ItemContentProviderTest {

// FOR DATA

private var mContentResolver: ContentResolver? = null

@Before

fun setUp() {

Room.inMemoryDatabaseBuilder(InstrumentationRegistry.getContext(),

SaveMyTripDatabase::class.java)

.allowMainThreadQueries()

.build()

mContentResolver = InstrumentationRegistry.getContext().contentResolver

}

@Test

fun getItemsWhenNoItemInserted() {

val cursor = mContentResolver!!.query(ContentUris.withAppendedId(ItemContentProvider.URI\_ITEM, USER\_ID), null, null, null, null)

assertThat(cursor, notNullValue())

assertThat(cursor!!.count, `is`(0))

cursor.close()

}

@Test

fun insertAndGetItem() {

// BEFORE : Adding demo item

val userUri = mContentResolver!!.insert(ItemContentProvider.URI\_ITEM, generateItem())

// TEST

val cursor = mContentResolver!!.query(ContentUris.withAppendedId(ItemContentProvider.URI\_ITEM, USER\_ID), null, null, null, null)

assertThat(cursor, notNullValue())

assertThat(cursor!!.count, `is`(1))

assertThat(cursor.moveToFirst(), `is`(true))

assertThat(cursor.getString(cursor.getColumnIndexOrThrow("text")), `is`("Visite cet endroit de rêve !"))

}

// ---

private fun generateItem(): ContentValues {

val values = ContentValues()

values.put("text", "Visite cet endroit de rêve !")

values.put("category", "0")

values.put("isSelected", "false")

values.put("userId", "1")

return values

}

companion object {

// DATA SET FOR TEST

private val USER\_ID: Long = 1

}

}

**Explanations:** We've created an instrumented test that will play the role of the outside app wishing to access our application’s data. For this, we used the class [**ContentResolver**](https://developer.android.com/reference/android/content/ContentResolver.html), taking a URI as input to communicate with our **ContentProvider** by calling its various methods.

Now let’s run the test. Once it’s done, if you start your app, you should see that a new "thing to do" has been added to your list... by your test... via your ContentProvider! :)

Pretty neat, isn't it? Our test played the role of the outside app and **has added real data** within our SQLite database!

To summarize, be aware that projects which implement the **ContentProvider** (or to be specific, that need to expose their data) are quite rare, in particular because of how complex this class is to use. But all the same, it’s important to see it at least once. ;)

### **Tips & Tricks to improve the quality of your code**

Your mini-app SaveMyTrip is now complete, and you’re really quite satisfied with it. So much so that you recently launched it on the Play Store... and it was a hit! :soleil: Now you need to hire several developers in order to **maintain** and **upgrade** it.

I’m beyond delighted. Of course, working in teams isn’t easy... especially since I need to **constantly check** that my employees aren’t breaking all the code that has already been created! :euh:

Ah! Well, this can easily be avoided, thanks to [**continuous integration**](https://fr.wikipedia.org/wiki/Intégration_continue)**!** :)

I have absolutely no idea what that means...

Yes, it seemed that way... ;) Here’s a definition adapted from Wikipedia, which summarizes this concept well:

Continuous integration is a **set of practices** used in software engineering consisting of **checking each time** the source code is edited that the changes don't result in any [**regressions**](https://fr.wikipedia.org/wiki/Test_de_régression) in the application under development.

The main purpose of this practice is to **detect** integration problems **early** in development. In addition, it allows you to **automate** the execution of test suites and to see the evolution of the development of the software.

In fact, currently, you’ve already created **a few tests** that "protect" your code to some extent, and which will enable you to continue development in relative peace. However, when you’re working on a team, an error quickly arises: A developer who modifies a feature and **doesn’t run the tests**, who then publishes your app for release... **what a disaster!** A regression occurs, something that you would’ve been able to avoid if the tests had been run earlier, in an automated way.

Thanks to continuous integration, you’ll be able to **automate this process** quite easily... ;) For that reason, in this last chapter, I’ll be showing you how to automate the running your tests with [Travis-ci](https://travis-ci.org/), which is free software for continuous integration, combined with [Codecov](https://codecov.io/), software for generating [code coverage](https://en.wikipedia.org/wiki/Code_coverage) reports.

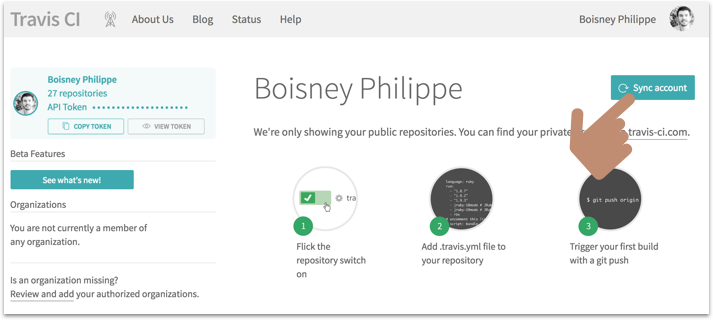
#### **Installing Travis**

First, create an account on the [Travis website](https://travis-ci.org/) using your **Github** login details.

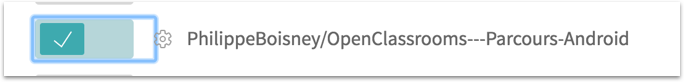
Wait, why does Travis need my Github login? :o

For the simple reason that Travis is **heavily integrated** with Github! In fact, as soon as you are going to **send a commit** on your Github repo, Travis will automatically launch the tests present inside your project in order to alert you if the commit in question involves any problems or not... :)

Once you’ve logged in to Travis, the following page appears:

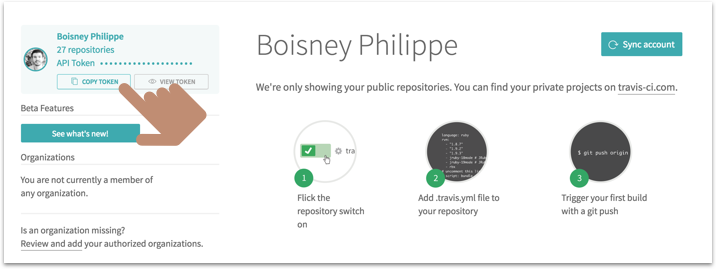
Sync Travis with Github

Click on the "**Sync account**" button so that Travis can retrieve a list of your repos. Once this is done, we’ll **enable Travis** on the repo containing the source code of our **SaveMyTrip** application:

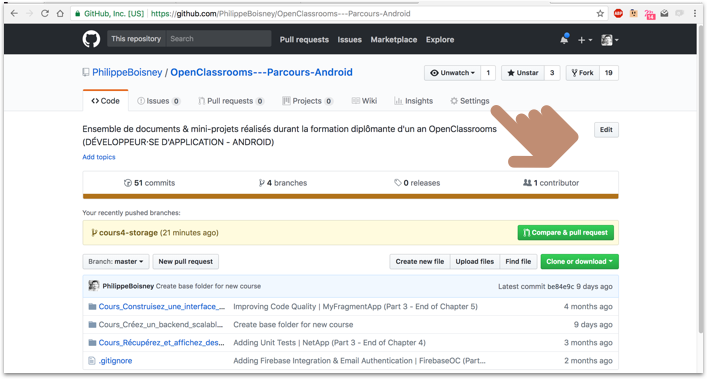
Activating the repository containing our app SaveMyTrip

Your Github repo absolutely **must** be in **public** mode if you want to use the **free** version of Travis.

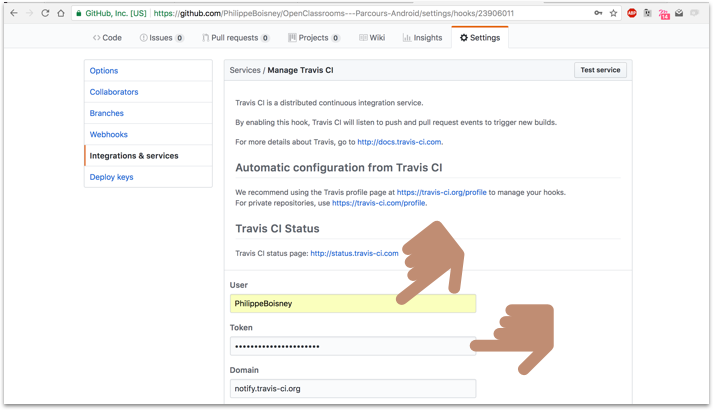
Now, retrieve the Travis token located at the top left of the Travis administration page by clicking on the button "**Copy Token**":

Retrieving the Travis token

Next, head to the Github repo containing the source code of your mini-app SaveMyTrip, and click on the button "**Settings**":

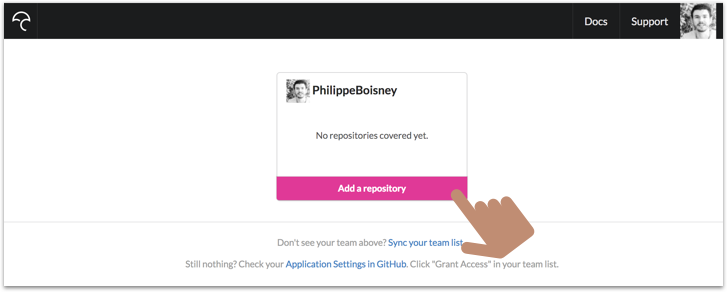
Accessing a Github repository’s settings

Then click on "**Integrations & services**" then on "**Travis CI**", and enter your Github login into the "**User**" field and the token previously retrieved from Travis in the "**Token**" field:

Integrating Travis into a Github repo

Next, save by clicking on the "**Update service**” button. Now, we’re going to move on to configuring a service allowing us to automatically generate [code coverage](https://fr.wikipedia.org/wiki/Couverture_de_code) **reports** so that we can accurately measure the **percentage** of our code covered by our tests. This service is called [Codecov](https://codecov.io/).

Log in, and click on the button "**Add a repository**":

Add a repository to Codecov

Select the repository containing the source code of your mini-app **SaveMyTrip,** and that’s it. **:)**No need to configure anything on this end!

#### **Configuring the Android app**

Now we’re going to have to configure our mini-app SaveMyTrip so that it can generate code coverage reports for each of its builds. For this, we’re going to install the library [**Jacoco-Android**](https://github.com/arturdm/jacoco-android-gradle-plugin).

**File build.gradle on your project level:**

buildscript {

repositories {

google()

jcenter()

}

dependencies {

...

classpath 'com.dicedmelon.gradle:jacoco-android:0.1.1'

}

}

**File build.gradle on your app level:**

apply plugin: 'jacoco-android'

android {

...

buildTypes {

...

debug {

testCoverageEnabled true

}

}

...

}

And there you go! Jacoco is installed. We also remembered to enable the **generating of coverage reports** thanks to the line testCoverageEnabled true . We will now be able to configure Travis, creating the following **configuration file** at the root of our Github repository: .travis.yml

**File .travis.yml:**

language: android

jdk: oraclejdk8

env:

global:

- ANDROID\_API\_LEVEL=27

- EMULATOR\_API\_LEVEL=22

- ANDROID\_BUILD\_TOOLS\_VERSION=27.0.3

- ANDROID\_ABI=armeabi-v7a

android:

components:

- build-tools-$ANDROID\_BUILD\_TOOLS\_VERSION

- android-$EMULATOR\_API\_LEVEL

- sys-img-armeabi-v7a-android-$EMULATOR\_API\_LEVEL

before\_install:

- yes | sdkmanager "platforms;android-27"

before\_script:

- chmod +x gradlew

- ./gradlew build jacocoTestReport assembleAndroidTest

- echo no | android create avd --force -n test -t android-$EMULATOR\_API\_LEVEL --abi $ANDROID\_ABI

- emulator -avd test -no-skin -no-audio -no-window &

- android-wait-for-emulator

- ./gradlew connectedCheck

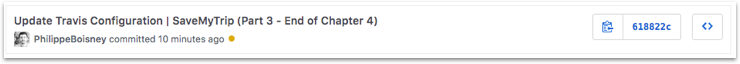
after\_success:

- bash <(curl -s https://codecov.io/bash)

**Explanations:** This file will be read by the service Travis directly from your Github repository. This is also why you absolutely must place it **at the root** of the repo.

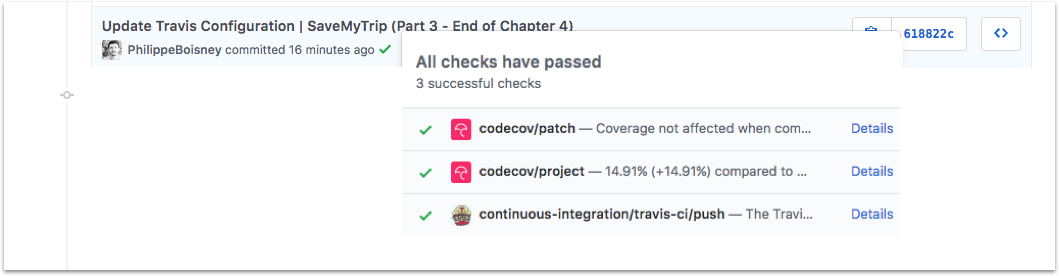
You’ll find more information about all of this file’s settings at the [**Travis website**](https://docs.travis-ci.com/user/languages/android/). To summarize, Travis will automatically launch the tests of our project on a virtual emulator using the command ./gradlew build jacocoTestReport assembleAndroidTest. Once they are completed, a **code coverage** **report** will be sent to the site Codecov.io owing to the parameter after\_success and the bash command below.

Once this file is completed, **perform a commit** of your Android project on Github, so that Travis can run. You should see, from the Github web interface, the image corresponding to your commit:

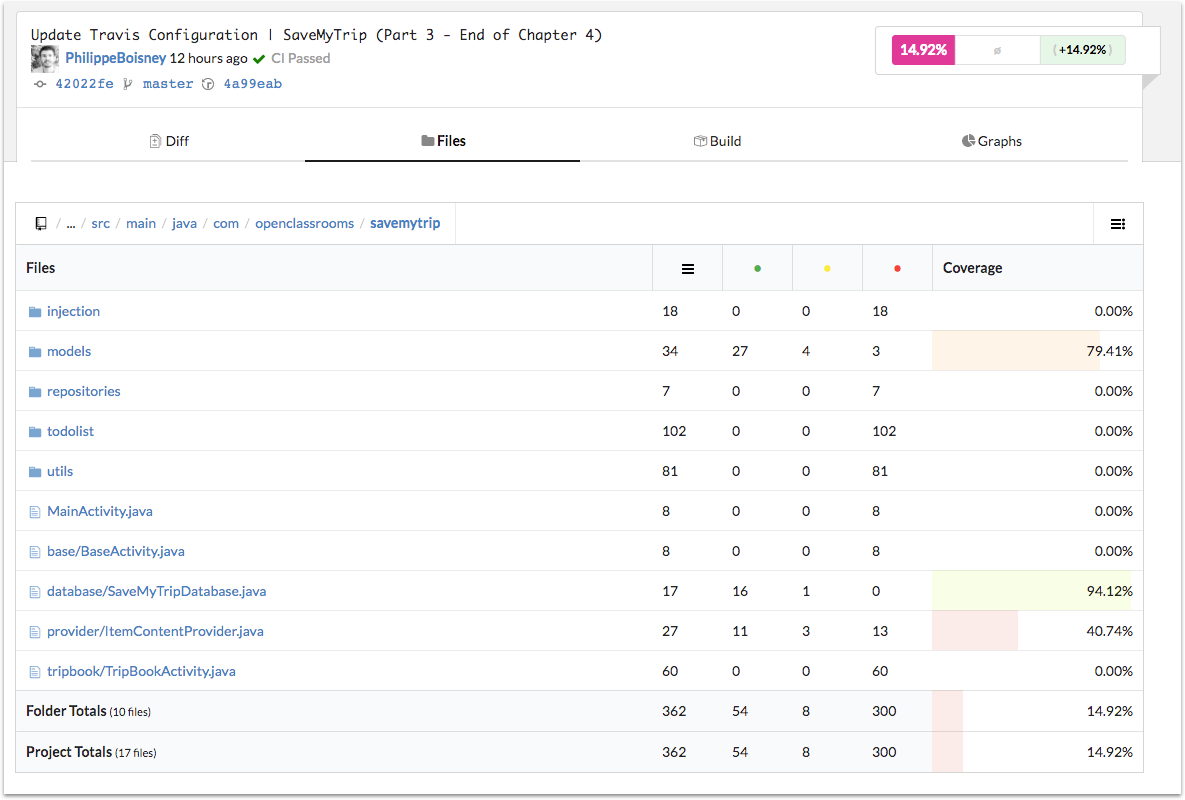
The service Travis is run on your commit...

The small **orange disc** indicates that Travis is currently analyzing your code and running all your tests. This may take a bit of time, usually a quarter-hour, because Travis must launch a virtual emulator in order to run your instrumented tests... ;)

As soon as Travis has finished running your tests, the disc will change to **green** or **red**, depending on whether they **succeeded** or **failed**.

Test results

Furthermore, if you click on the service **codecov**, you’ll be able to view the entire code coverage report... :)

Code coverage report

For example, you can find mine publicly [at this link](https://codecov.io/gh/PhilippeBoisney/OpenClassrooms---Parcours-Android/tree/42022fef885570fe4b403f8c8a2f415f984c2046/Cours_Gerez_vos_donnees_localement_pour_avoir_une_application_100_offline/Partie1_2_3/SaveMyTrip/app/src/main/java/com/openclassrooms/savemytrip) as well as the running of my tests [on Travis](https://travis-ci.org/PhilippeBoisney/OpenClassrooms---Parcours-Android/builds/353509728?utm_source=github_status&utm_medium=notification). OK, as you can see, our tests cover approximately **15%** of our code... That’s good... but we can do better! **>_<**

Besides, the goal isn’t necessarily to reach 100%, but rather to cover all the **business features** of our code through testing.

#### **Conclusion**

And now the course is complete! I hope you enjoyed it and understood it well. :) You might not realize it, but you just took a big step forward in developing for Android! We saw a great deal of **fairly advanced concepts**, so take the time to absorb them.

We saw how to **store data** in different **storage spaces** on Android within a text **file** and how to expose it publicly in a secure manner. We also studied how to store structured data using a **SQLite database**, using the library **Room**. Finally, we looked into how to improve the quality of our Android apps with **Architecture Components** and **continuous integration**.

And as always, remember to **practice**, **practice** and **practice** some more, so that coding becomes as natural as breathing... :p

See you soon in another course!

Get the completed code for the **SaveMyTrip** app used in this course [from Github](https://github.com/PhilippeBoisney/OpenClassrooms---Parcours-Android/tree/master/Cours_Gerez_vos_donnees_localement_pour_avoir_une_application_100_offline/Partie1_2_3/SaveMyTrip).

**Description**

Welcome to this peer graded activity!

Your mission is to improve the application SaveMyTrip, which is available for download here. Your task is to spruce up the your "to do" list by adding image support. It should look like this:

**Expected result**

Here's what you have to do:

Add a button allowing the user to retrieve images from their personal photo gallery (which will use external storage). This image will show up in place of the button. (In the example above, this would be the transition from Screen 1 to Screen 2.)

Once the user clicks on the big green "Add" button at the bottom of the screen, a new "to do" should be added to the list. It should contain the image previously selected by the user, as well as a "share" button. (In the example above, this would be screen 3.)

The "Share" button should only show up if an image is present, because this button allows the user to share this image. You should therefore also implement a function allowing the user to share the image from their "to do" list when they click on the button.

The "Share" functionality should allow your user to share to one or more options: email, social media, etc.

You don't need to use the class FileProvider to share the image.

The goal is to make a functional application which looks like the expected result above.

Good luck!

**Deliverable**

You'll give the complete repository of your Android project, compressed in a ZIP format.

Avoid including the build/ files so that you can keep your zip light.

**Evaluated skills**

This activity will allow you to put skills below into practice. Your activity will evaluated by your peers according to the stated evaluation criteria for each skill.

Implement a permanent architecture in Android

Evaluation criteria

In order to validate this skill, you will have to make changes to an existing application.

Send your activity