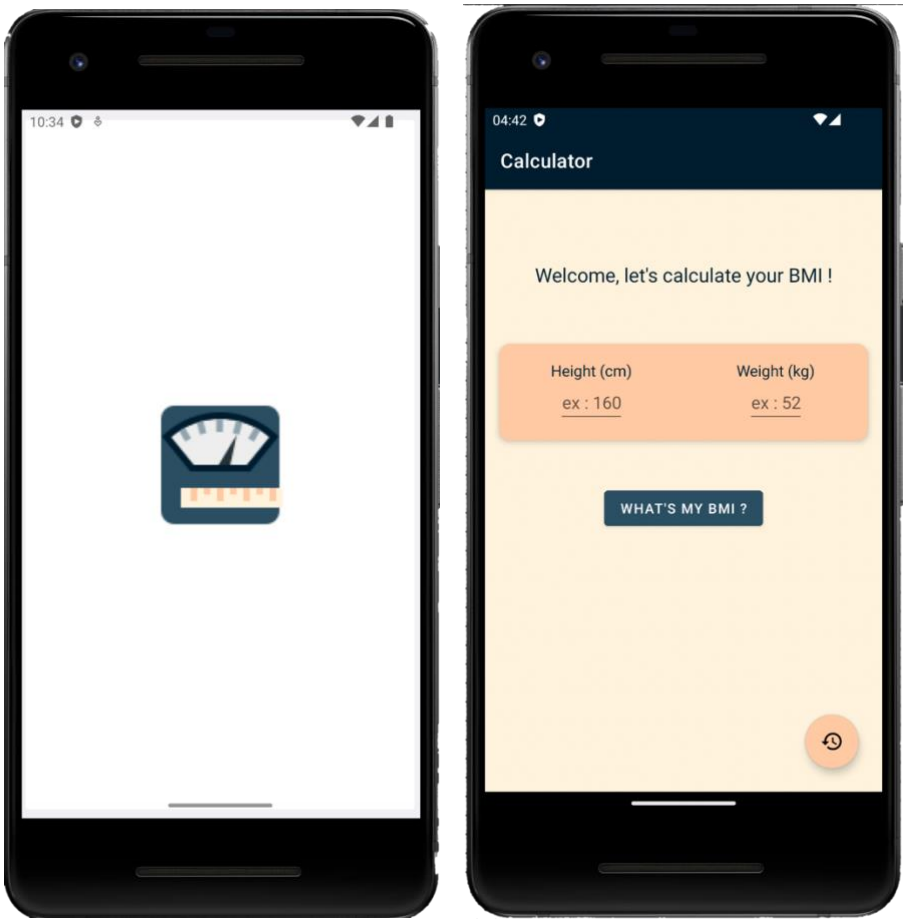
UiT The Arctic University of Norway



My BMI App – Upgraded version

MANDATORY ASSIGNMENT 3 - REPORT

Lise GAUTHIER | INF-2710 mHealth

Table of contents

Description of the application

VERSION 1.0

"My BMI App" is an Android application that allows users to enter their height and weight, and calculate their Body Mass Index (BMI) by clicking on a button (see *Figure 1*). When the user clicks on the "What's my BMI ?" button, he obtains his BMI value as well as his nutritional status. For better visualization, this result is associated with a circular progress bar that fills in and colors according to the BMI value (see *Figure 2*). The result is shown in *Figure 3*.

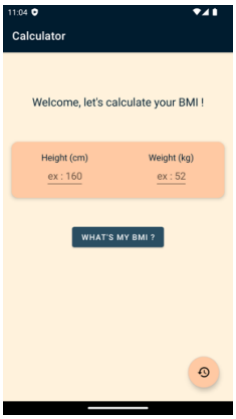


Figure 1: Screenshot of the page to calculate the BMI

BMI	(kg/m2)
Underweight (Severe thinness)	< 16.0
Underweight (Moderate thinness)	16.0 – 16.9
Underweight (Mild thinness)	17.0 – 18.4
Normal range	18.5 – 24.9
Overweight (Pre-obese)	25.0 – 29.9
Obese (Class I)	30.0 – 34.9
Obese (Class II)	35.0 – 39.9
Obese (Class III)	≥ 40.0

Figure 2 : BMI Classification

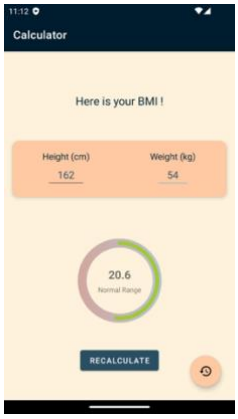


Figure 3 : Screenshot of the page displaying the BMI

The user can recalculate a BMI by clicking on the "Recalculate" button. He is then redirected to the interface presented in *Figure 1* with the only difference that the input fields keep the last values entered by the user (see *Figure 4*). These fields are also pre-filled each time the application is restarted.

Each time a BMI is calculated, the application records it and adds it to the user's history. To access the history, the user must click on the floating button, located at the bottom right of the page. The history is empty when the user uses the application for the first time (see *Figure 5*), but each time the application is launched, it is retrieved and updated (see *Figure 6*).

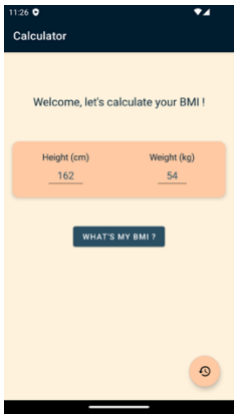


Figure 4 : Screenshot of the page displayed after clicking on "Recalculate"

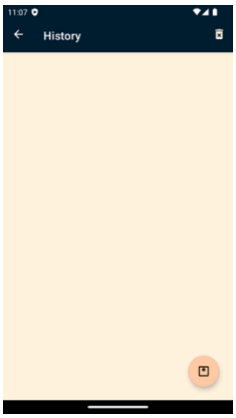


Figure 5 : Screenshot of the empty history

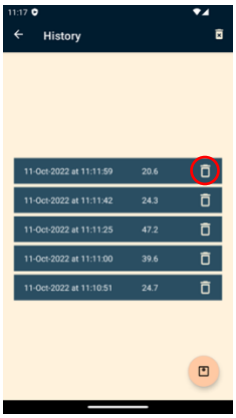


Figure 6 : Screenshot of the history

Once the BMIs are added to the history, the user can delete them one by one by clicking on the trash can icon (see red circle on *Figure 6*). In this case, a confirmation window will open to confirm the user's choice (see *Figure 7*). If the user clicks on "Yes", the BMI is deleted and a Toast is displayed to confirm the deletion (see *Figure 8*).

The user can also empty the entire history by clicking on the trash can icon in the upper right corner. As in the previous case, a confirmation window will appear (see *Figure 9*) and if the user confirms, the history is completely emptied and a Toast is displayed to confirm the deletion (see *Figure 10*).

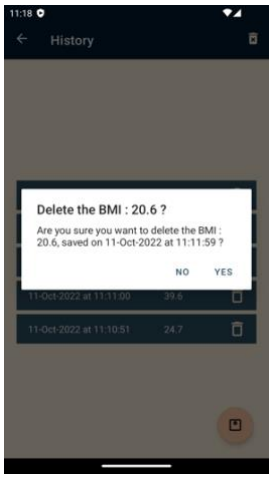


Figure 7 : Screenshot of the confirmation window for deleting a BMI

Figure 8 : Screenshot of the history, after deleting a BMI

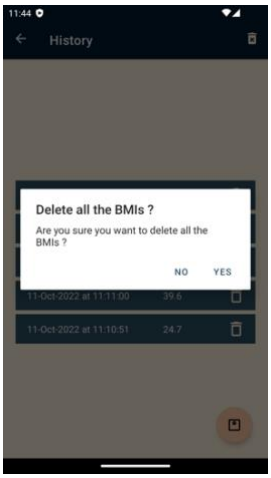


Figure 9 : Screenshot of the confirmation window for deleting all BMIs

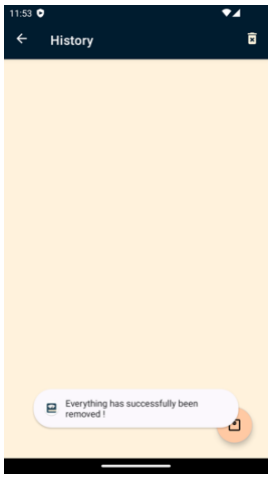


Figure 10 Screenshot of the history, after deleting all BMIs

To return to the "Calculator" page, the user can click on either the floating button at the bottom right or the arrow at the top left.

VERSION 2.0

One of the most important improvements in this new version is the addition of user profiles. Thus, as shown in *Figure 11* and *Figure 12*, several people can create their profile and use the app to calculate their BMI and access their own history. It is also possible to delete the users one by one and all at the same time, by clicking on the trash icons. What's more, when creating a profile, the user's height is required so that the field is pre-filled for each BMI calculation (see *Figure 13*). However, if a new height is input by the user, it modifies his current height value.

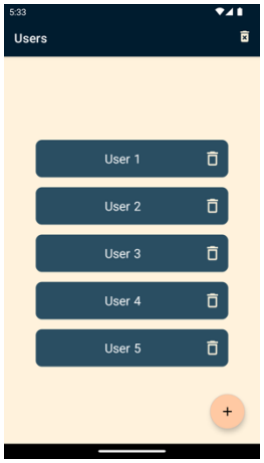


Figure 11 : Screenshot of the interface displaying the list of users

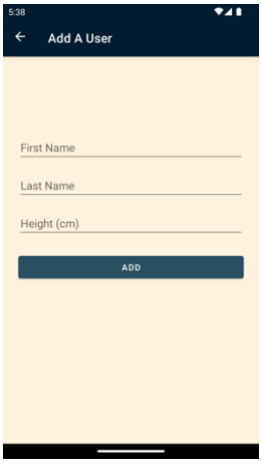


Figure 12 : Form to add a profile

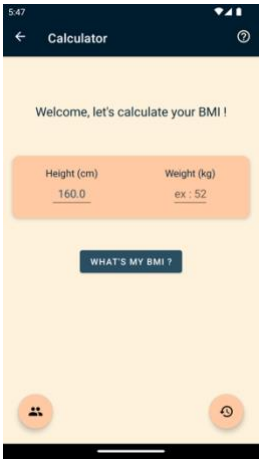


Figure 13 : Pre-filling of the height input field

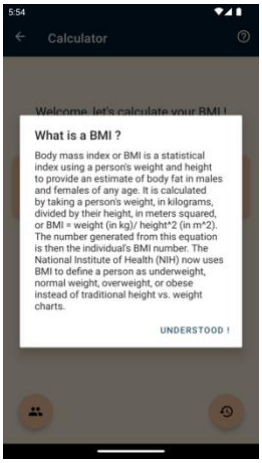


Figure 14 : Introducing the BMI topic in the app

This new version of the application introduces the BMI topic. To access it, the user must go to the calculator interface and click on the "help" icon in the upper right corner (see *Figure 14*).

Changes have also been made from the first version. For example, when the user clicks on the "Recalculate" button (see *Figure 3*), instead of being redirected to the first calculator interface (*Figure 4*), a new BMI is calculated from the values present in the input fields.

In the history, as shown in *Figure 15*, the user can now access the weight and height associated with each BMI. In addition, each BMI record is displayed in a certain color, depending on the BMI value.

Figure 15 : New history version

The functionality of pre-filling fields with the latest values has been removed. In this new version, only the height field is pre-filled with the user's current height. Indeed, it did not seem relevant to save the last weight entered because it is likely to change regularly.

Features implemented

VERSION 1.0

As presented in the previous section, the mobile application offers many features. To satisfy the instructions of the assignment, it calculates the BMI from a weight and a height entered by the user. It saves the calculated BMIs as well as their recording date to display them in a history. It is possible to switch from the "Calculator" page to the "History" page and vice versa, thanks to a floating button.

I also chose to implement additional features to make this application more complete and user-friendly. First, in order for the user to have a more visual overview of his BMI, I generated a circular progress bar whose progression level and color adapt according to the BMI value, as shown in *Figure 2*. For example, for a BMI in "Normal Range", the progress bar will be green and half filled (see *Figure 16*). The higher the BMI value is, the more the progress bar fills up and becomes red. Conversely, the lower the BMI is, the less the progress bar fills up and the more it tends to turn blue (see *Figure 17*).

Figure 16 : Example of a circular progress bar for a BMI in "Normal Range

Figure 17 : Colors depending on BMI values

Figure 18 : Application's logo

The mobile application keeps the last data entered by the user, even if the user quits the application. Thus, the saved data pre-fills the weight and height fields when the user re-launches the application. This saves time when the user wants to calculate his BMI again, since his height is unlikely to change.

I also added the functionality to delete BMIs. Thus, it is possible to delete the records one by one, or all at once. This feature seemed interesting to implement because it is possible that the user makes a mistake when entering his height or weight and wants to delete the wrong BMI from his history. Finally, for a design issue, I customized the icon of the app to be consistent with the features and the graphic charter (see *Figure 18*).

VERSION 2.0

As mentioned before, new features have been implemented in this second version. Indeed, to make this application even more complete, it is now possible to use several profiles on the application. Each user, filling in his first name, last name and height, can access the calculator and his personal history. For each BMI calculation, the user only needs to enter his weight, as his height is pre-registered when the profile is created. If the user modifies the height field that has already been pre-filled, it updates the height that was entered when the profile was created. It is also possible to delete one or more profiles.

This new version also allows the user to have access to information. Indeed, from the calculator interface, he can learn what exactly is the Body Mass Index. To improve the user's understanding, the height and weight are now displayed in each record of the history. In addition, each record is displayed on a colored background, which adapts according to the BMI value. This feature allows the user to quickly visualize the evolution of his BMI in the history.

Finally, the functionality to pre-fill fields with the last values entered has been modified. Only the height field is pre-filled with the user's current height, which is filled in when creating a user profile or updating the height, when the user changes the height field.

Development process

PROJECT ARCHITECTURE

In any project, it is important to adopt a good project architecture. As shown in the figures below, my project is composed of a main file **MainActivity** and several packages, each associated with a specific functionality (see *Figure 19*). The "**data**" package contains all the files necessary to set up and use the Database, concerning the storage of BMIs and users. The "**ui**" package corresponds to the files used to display the user interface. An XML file is associated to each of these files, stored in the "layout" package, a little lower in the tree structure (see *Figure 20*). The "**ui**" package is subdivided into three subfolders, one for the "Calculator" view, one for the "History", and one for “User Interface”.

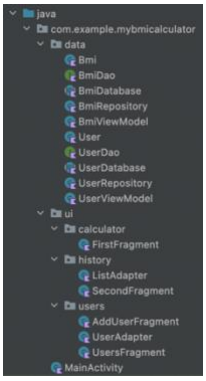


Figure 19 : Project architecture - Kotlin files

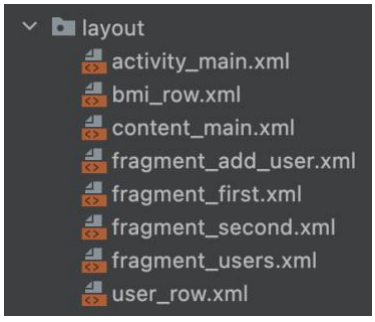


Figure 20 : Project architecture - XML files

All the files/methods/classes mentioned hereafter are available in the project folder, attached to this report, if needed.

USER INTERFACE DESIGN

Concerning the user interface design, I used the Basic Activity Template proposed by Android Studio, which I stylized to make the application more attractive and aesthetic. My application consists of four interfaces. The first one displays the BMI calculator (corresponding to the FirstFragment in the project architecture), the second one shows the history (SecondFragment), the third one contains a form to add a user (AddUserFragment) and the last one displays the list of all users (UsersFragment).

CALCULATOR FEATURES

User input management

For the BMI calculation, it is important to check that the values entered by the user in the input fields are correct. For this purpose, I have created a **checkInput()** function which first checks if the fields are not empty. This function then checks that the values entered by the user are positive numbers. After coding this function, I realized that by restricting the type of data that can be entered in the **EditText** view, to "*decimal number unsigned*", part of my function was useless.

Calculation and display of BMI

If the values entered by the user are correct, the BMI is calculated using the formula : $\frac{weight(kg)}{height(m)^2}$.

To display the result, in addition to displaying the BMI value and the nutritional status, I decided to implement a **CircularProgressBar** view whose color and filling vary according to the BMI value. As explained earlier, the higher the BMI, the more red and filled the progress bar is; and conversely, the lower the BMI, the less filled the progress bar is and closer to blue. To do this, I had to install a new dependency in the *build.gradle* file and document myself in order to understand how it works so I could adapt it to my needs.

Backup of last input data

Concerning the storage of the last data entered by the user, I did some research and I found that the **SharedPreferences** interface was well appropriate. Indeed, a SharedPreferences object allows to store and retrieve simple values from a file containing key-value pairs. So, I created two functions that use this interface, one to save the last inputs, and the other to load them.

In the second version of the application, the input fields are no longer pre-filled with the last saved values. Thus, the two methods using SharedPreferences, previously mentioned, are no longer used. From now on, only the height field is pre-filled with the current height of the user, stored in the database.

Pre-fill of the height field

To pre-fill the height field, I created the function **getCurrentHeight(id)** in the DAO **UserDao** (see *Figure 21*), which retrieves the height of the user whose id is passed in parameter. I also created the function **updateHeightBy(id, height)**, to update the height if a new height is input.

```
//query to retrieve the the current height of a user
@Query("SELECT currentHeight FROM user_table WHERE userId=:userId")
fun getCurrentHeight(userId: Int): Double

//query to update the user's height
@Query("UPDATE user_table SET currentHeight=:height WHERE userId=:id")
fun updateHeightBy(id: Int, height: Double)
```

Figure 21 : getCurrentHeight and updateHeightBy functions in UserDao

Then we add these functions in the Repository **UserRepository** and the ViewModel **UserViewModel**. The operation of the DAO, Repository and View Model is described later.

Since these functions take the id of a user as a parameter, it is necessary to be able to access the id of the current user. To do this, in the navigation between the different fragments, we add the attribute currentUser (which will be shared between the different fragments) and we create a variable of type navArgs (see *Figure 22*), which allows access to this currentUser. This variable retrieves the user selected in the UserFragment and makes it accessible in FirstFragment.

```
//to retrieve the current user we have selected and pre-fill his height in the height input
private val args by navArgs<FirstFragmentArgs>()
```

Figure 22 : Instanciation of a navArgs variable

We can then retrieve and pre-fill the currentHeight of the currentUser in the database, via the method getCurrentHeight taking as parameter the id of the currentUser, retrieved by the navArgs variable (see *Figure 23*).

```
//pre-fill the height input
val currentHeight = mUserViewModel.getCurrentHeight(args.currentUser.userId)
view.height_input.setText(currentHeight.toString())
```

Figure 23 : Use of getCurrentHeight function, using the navArgs variable

The updateHeight() function is only called if the value entered by the user is different from the one stored in the database (see *Figure 24*).

```
private fun updateHeight(height: Double) {

    //we update if the height is not empty and if it is different from the current height
    if (args.currentUser.height != height) {
        // Update Current Height
        mUserViewModel.updateHeight(args.currentUser.userId, height)
        Toast.makeText(requireContext(), text: "Height updated Successfully!", Toast.LENGTH_SHORT).show()
    }
}
```

Figure 24 : updateHeight function

HISTORY FEATURES

To implement the history, I needed to access a local database to save the different BMI values calculated. After some research, I decided to use the Room library associated to ViewModel because it allows to create easily a database and to perform CRUD operations (Create, Read, Update, Delete). Moreover, my choice was quickly oriented towards Room because I had already manipulated the repository and DAO principle with the Spring framework.

Creation of the Database

The first step is to create an entity class **Bmi**, containing the attributes *bmiValue* and *dateAdded*. To implement all the features of the second version of the app, it is necessary to add more attributes such as the *heightValue*, the *weightValue*, and the *userOwnerId* (which represents the id of the user to which the BMI belongs). Then, we create a Data Access Object (DAO) **bmiDao** which implements functions that directly execute SQL queries in the database. For example, the function shown in *Figure 25* allows us to retrieve all the BMIs, ordered by decreasing date of addition.

```
//query to display all BMIs
@Query("SELECT * FROM bmi_table ORDER BY dateAdded DESC")
fun  readAllBmi(): LiveData<List<Bmi>>
```

Figure 25 : Example of a function created in bmiDao

We then create the file **BmiDatabase** which generates the database, and a repository **BmiRepository**, which contains functions calling other functions defined in the DAO.

Finally, we create a ViewModel **BmiViewModel** which makes the link between the repository and the UI (User Interface). It survives configuration changes, provide data to the UI from the repository. The mechanism of access to the database from the UI is summarized in *Figure 26*.

Figure 26 : Diagram of the access to the Database from the UI

Add to database feature

To add a BMI to the database, I created the function **insertDataToDatabase()** in FirstFragment. This function retrieves the BMI calculated from the values entered by the user and formats the date added as a String so that it can be added to the database. At first, in the **Bmi** entity, I had defined the *dateAdded* attribute of type Date but I quickly realized that it was easier to manipulate attributes of type String. While testing my function, I also noticed that the results were well ordered by descending date of addition (as implemented in the DAO) but it did not take into account the hour of addition so my results were not displayed correctly. To fix this problem, when formatting the date, I specified that the hours were also taken into account.

Database reading feature

To display the contents of the database, I created a **row_item** layout that is instantiated for each row of the database and contains the *bmiValue* and *dateAdded* attributes of the row in question.

These layouts are contained in the **ListAdapter** fragment, of type *RecyclerView*, which contains the dynamic list of BMIs in the database.

To associate the UI with the data, we add the RecyclerView in SecondFragment and instantiate a ViewModel that executes the **readAllData()** method. This method performs an SQL query in the database through the repository and DAO. The returned results are added to the ListAdapter via the method **setData(bmi).**

Deleting to database feature

The user has the possibility to delete the records 1 by 1 or to delete them all at the same time. At the code level, the process is the same with the only difference that it is executed in two different fragments/components. For the individual deletion, it is implemented in **ListFragment** and for the deletion of all BMIs, it is implemented in **SecondFragment**.

When clicking on a garbage can icon, we call the **deleteAllBmis***()* method (or ***deleteBmi()***), which calls the *deleteAllBmis()* method of the ViewModel, which calls the *deleteAllBmis()* method of the repository, which itself calls *deleteAllBmis()* of the DAO, which executes the SQL query: "*DELETE FROM bmi_table*".

PROFILE MANAGEMENT FEATURES

For the users management, I proceeded in the same way as for the history. I created a UsersFragment that displays the list of added users, thanks to a Recycler View. Unlike the history, I created another fragment AddUserFragment, which contains a form for creating a new profile. I then linked all the fragments together by setting up navigations and making the home page the UsersFragment.

Creation of the database

In order to store the users in the database, I created a **User** entity containing the *first name*, *last name* and *height* attributes. The height attribute is used to save the current height of the user and, as it is not likely to change often, it is used to pre-fill the height field of the BMI calculator.

I then created the **UserDao**, **UserDatabase**, **UserRepository** and the **UserViewModel**, implementing many functions such as adding, removing a user, displaying all users, etc.

Displaying and deleting users

To display the list of all users, I proceeded as for the history. I created a **UserAdapter** and a **user_row** layout that will dynamically fill the Recycler View for each user.

As for the history, it is possible to delete users one by one, or all at once. To do this, I created two functions **deleteUser()** (used in UserAdapter) and **deteleAllUsers()** (in UsersFragment), both of which call functions defined in UserDao.

Associations of Users with their BMIs

To associate users with their BMIs, I used the navigation attributes. For each fragment that requires the currentUser, we define a variable of type navArgs. When redirecting fragments, we get the currentUser thanks to this variable and we pass it as a parameter to the redirection function, so that the next fragment can execute the functions that require the id or the height of the currentUser. These redirection functions are present on the *Figure 27*, *Figure* 28 and *Figure* 29.

```
//share the current item between the Users view and the calculator
holder.itemView.userLayout.setOnClickListener(){ it: View!
    val action = UsersFragmentDirections.actionUsersFragmentToFirstFragment(currentItem)
    holder.itemView.findNavController().navigate(action)
}
```

Figure 27 : Redirection function in UserAdapter, from UserAdapter to FirstFragment

```
//when we click on the right floating action button, the history view is displayed
view.findViewById<FloatingActionButton>(R.id.floatingActionButtonToHistory).setOnClickListener { it: View!
    val action = FirstFragmentDirections.actionFirstFragmentToSecondFragment(args.currentUser)
    findNavController().navigate(action)
}
```

Figure 28 : Redirection function in FirstFragment, from FirstFragment to SecondFragment

```
//share the current item between the history view and the calculator
val action = SecondFragmentDirections.actionSecondFragmentToFirstFragment(args.currentUserSecond)
findNavController().navigate(action)
```

Figure 29 : Redirection function in SecondFragment, from SecondFragment to FirstFragment

The mechanism of redirections and navArgs variables is summarized in the *Figure 30* below :

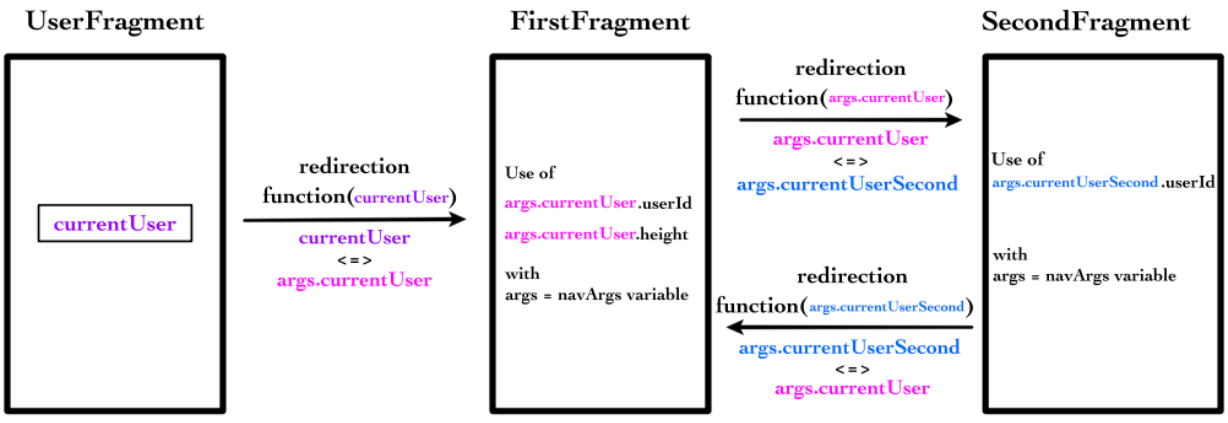


Figure 30 : Schema illustrating the redirections with navArgs variables in the app

Displaying and deleting user's BMIs

To display all the BMIs of a user, I used the readAllBmi() function (see *Figure 31*) to implement the **readBmiByUser(id)** function in BmisDao, as shown in the *Figure 32*. This function takes as parameter the id of the user whose history we want to display. I also implemented these functions in BmiRepository and BmiViewModel.

```
//query to display all BMIs
@Query("SELECT * FROM bmi_table ORDER BY dateAdded DESC")
fun  readAllBmi(): LiveData<List<Bmi>>
```

Figure 31 : readAllBmi function, not used in the second version of the app

```
//query to display the BMIs of a user
@Query("SELECT * FROM bmi_table WHERE userOwnerId = :id ORDER BY dateAdded DESC")
fun  readBmiByUser(id: Int): LiveData<List<Bmi>>
```

Figure 32: readBmiByUser function

For deletion, the process is the same. As shown in the *Figure 33*, I modified the deleteAllBmis() function of BmiDao to create the **deleteAllBmisByUser(id)** function, with id, the id of the user whose history we want to delete. I then implemented the function in the Repository and the View Model.

```
//query to delete all bmi
@Query("DELETE FROM bmi_table WHERE userOwnerId = :id")
suspend fun deleteAllBmiByUser(id: Int)
```

Figure 33 : deleteAllBmiByUser function

INTRODUCTION OF THE BMI TOPIC

To allow the user to learn more about the topic of BMI, I created a button in the navigation navbar of the calculator interface. To display the BMI information, I used an AlertDialog which is a kind of pop-up window. I chose this type of display because I found it more interactive than a simple fragment.

This pop-up window is opened with the **displayInformation()** function (see *Figure 34*), which is executed when the user clicks on the "help" icon of the navbar.

```
//function that displays information about BMI
private fun displayInformation() {
    //opens a window to confirm the deletion
    val builder = AlertDialog.Builder(context)
    builder.setPositiveButton( text: "Understood !"){_,_ ->
    }
    builder.setTitle("What is a BMI ?")
    builder.setMessage("Body mass index or BMI is a statistical index using a person's weight and height to provide " +
            "an estimate of body fat in males and females of any age. " +
            "It is calculated by taking a person's weight, in kilograms, divided by their height, in meters squared, " +
            "or BMI = weight (in kg)/ height^2 (in m^2). The number generated from this equation is then " +
            "the individual's BMI number. The National Institute of Health (NIH) now uses BMI to define a person " +
            "as underweight, normal weight, overweight, or obese instead of traditional height vs. weight charts. ")
    builder.create().show()
}
```

Figure 34 : displayInformation function

Difficulties and improvements to consider

VERSION 1.0

During the creation of this application, I had to face some difficulties. The main one was related to the fact that I was working with a brand-new framework. So, I trained myself thanks to tutorials but I had to solve many errors related to the execution of *build.gradle* files. Stackoverflow allowed me to solve them. Moreover, I had to restart my project several times because of GIT problems and problems with Fragments handling, as the first version of my application was based on a template I had not trained on. Another difficulty, mentioned before, was the manipulation of Date attributes in the database. To solve this problem, I formatted the Date objects in String.

Concerning the improvements, to improve the usability, it could be practical to recalculate automatically the BMI when pressing on the “Recalculate” button, instead of go back to the home interface. It could also be interesting to propose different profiles so that the application can be used by several people, and it could be relevant that the user enters a date when calculating the BMI, so that only one BMI can be saved per day. Finally, concerning the aesthetics, when designing the application, I had the idea to implement a Bottom Bar to switch between the calculator and the history. Due to lack of time and too many complications linked to the lack of knowledge of Kotlin, I resigned myself to use a floating button. It could be interesting to add this navigation mode in the future, to improve the user experience.

VERSION 2.0

In this new version, I tried to improve the usability and to make my application more complete. So, I implemented some features that I had mentioned in the improvements to consider in version 1.0. These improvements are the suppression of the redirection after the click on the "Recalculate" button, and the addition of profiles. Many other features, mentioned in this report, have been added.

I had much less difficulties to implement this new version because I understand and master better and better the Kotlin language and how to manipulate the Room Library.

Nevertheless, to improve the user experience, some features are still possible such as the possibility to record only one BMI per day by entering a date when calculating a BMI. It could also be interesting for users to customize their interface.

Finally, the application could go beyond the simple calculation and saving of BMIs, by nudging the user to practice a physical activity for example.