Habit tracker

Julián Camilo Montoya Rodríguez - 201615817 – jc.montoyar@uniandes.edu.co

Juan Sebastián Numpaque Roa – 201316637 – js.numpaque10@uniandes.edu.co

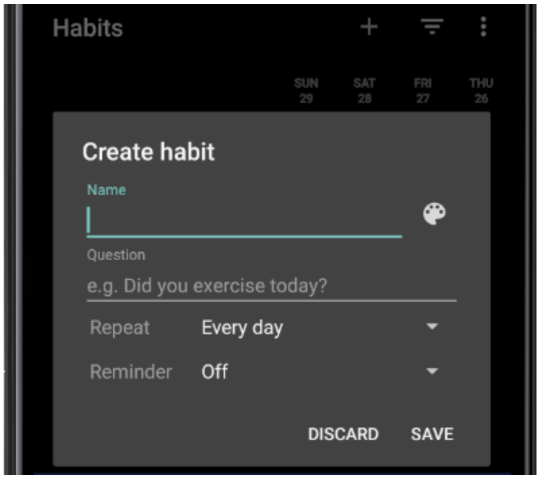
Contents

1. Introduction

The objective of this report is to make an evaluation of the open-source application Loop Habit Tracker over its technical aspects, such as performance, memory management, security, caching strategies, eventual connectivity and possible micro-optimizations that can be effected on its source code.

The profiling will be made on the android version of the application, which is mostly written in Kotlin. As of now, this application has over 40K reviews (average score of 4.5/5) and more than 1M downloads in the Google Play Store.

Its main purpose is to help its user to keep a track of their habits, whether it’s a habit that the user already does, or it’s a new habit that they want to start adding to their life. In the app the user creates the habits by assigning 3 main attributes: The name, a question to be asked to the user in order to keep track of the times they have completed the habit, and the days of the week in which the user wants to do the habit. Given that, the app notifies each day (of the selected ones) and asks the user whether if they completed the habit or not.



Create habit

Name

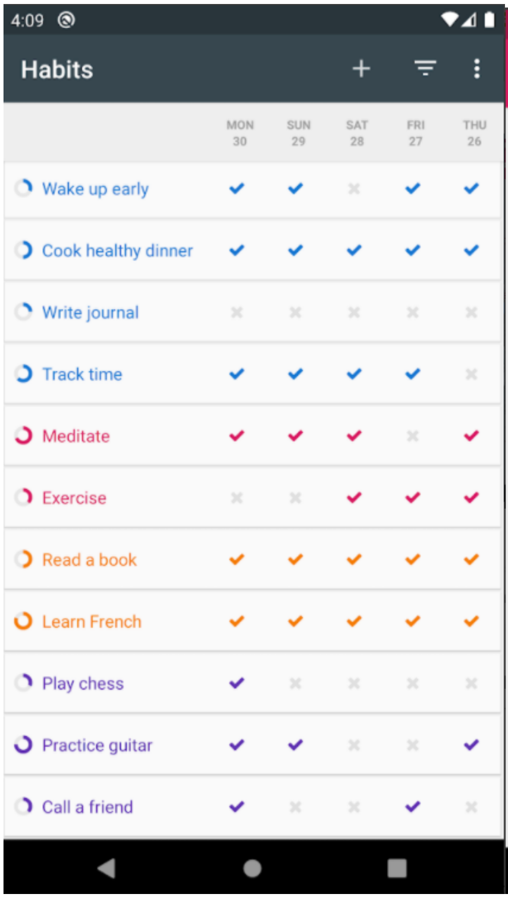
Question

e.g. Did you exercise today?

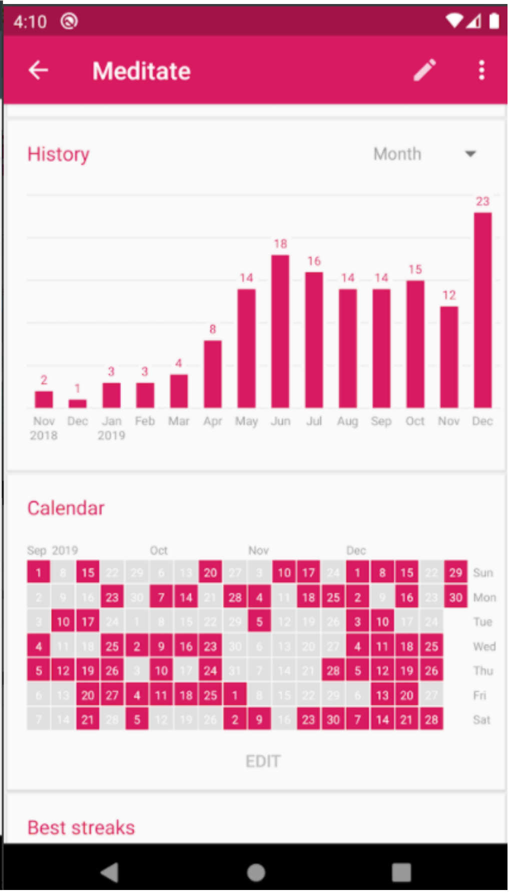
Repeat Every day

Reminder Off

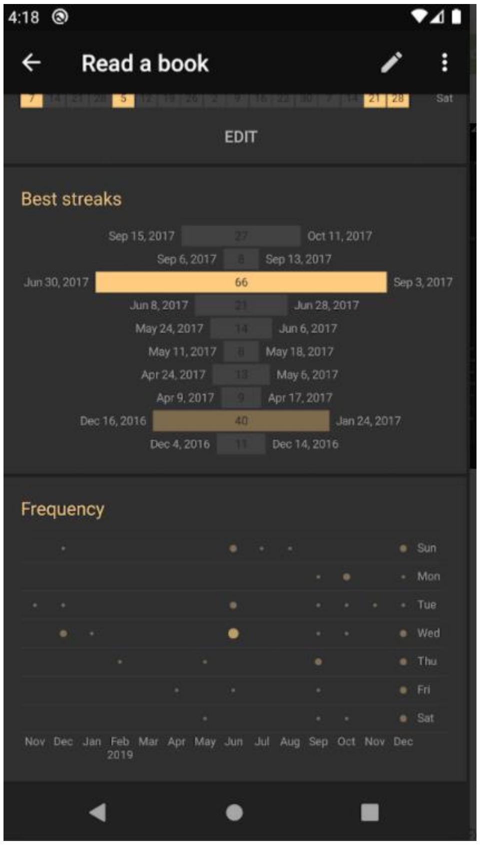
The app shows its full functionality when the user has used the app for a few weeks, and it shows for each habit a very detail dashboard related to the fulfillment of the habit, giving the user key data that can be used in order to understand how to improve in order to fulfill the habit.



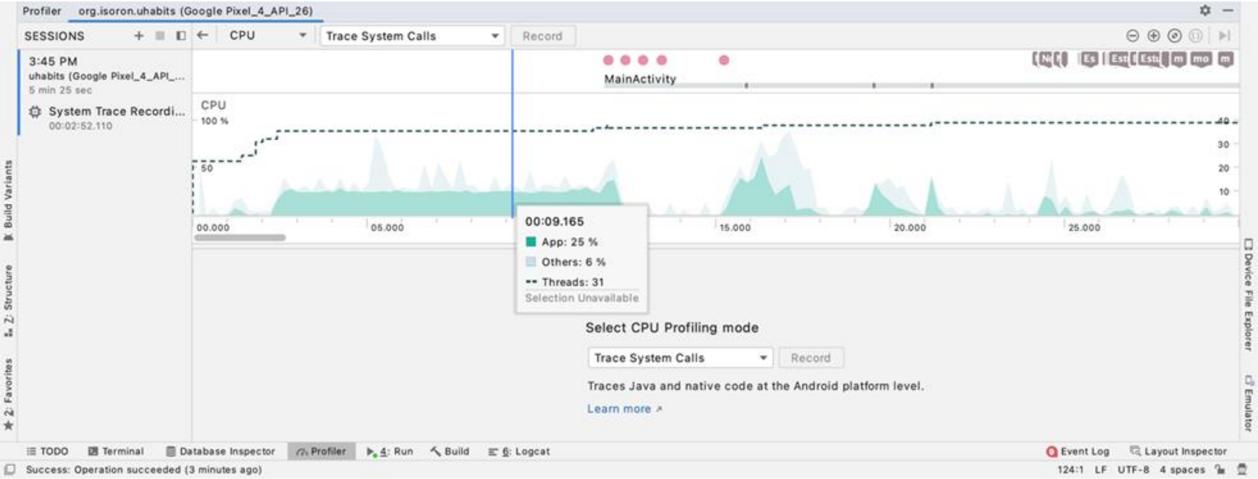
	MON 30	SUN 29	SAT 28	FRI 27	THU 26
Wake up early	✓	✓	✗	✓	✓
Cook healthy dinner	✓	✓	✓	✓	✓
Write journal	✗	✗	✗	✗	✗
Track time	✓	✓	✓	✓	✗
Meditate	✓	✓	✓	✗	✓
Exercise	✗	✗	✓	✓	✓
Read a book	✓	✓	✓	✓	✓
Learn French	✓	✓	✓	✓	✓
Play chess	✓	✗	✗	✗	✗
Practice guitar	✓	✓	✗	✗	✓
Call a friend	✓	✗	✗	✓	✗



Sep 2019					Oct				Nov				Dec					
1	8	**15**	22	29	6	13	**20**	27	3	**10**	**17**	24	**1**	**8**	**15**	22	**29**	Sun
2	9	16	**23**	30	**7**	**14**	21	**28**	**4**	11	**18**	**25**	**2**	9	**16**	23	**30**	Mon
3	**10**	**17**	24	1	8	15	22	29	**5**	12	19	26	**3**	**10**	17	24		Tue
4	11	18	**25**	**2**	**9**	**16**	**23**	30	6	13	20	27	**4**	**11**	**18**	**25**		Wed
5	**12**	**19**	**26**	3	**10**	17	**24**	31	7	14	21	**28**	**5**	**12**	**19**	**26**		Thu
6	13	**20**	**27**	**4**	**11**	**18**	**25**	**1**	8	15	22	29	6	**13**	**20**	27		Fri
7	14	**21**	28	**5**	12	19	26	**2**	**9**	16	**23**	**30**	**7**	**14**	**21**	**28**		Sat



2. Performance



00:09.165
🟩 App: 25 %
□ Others: 6 %
-- Threads: 31
Selection Unavailable

Select CPU Profiling mode

Trace System Calls ▾ Record

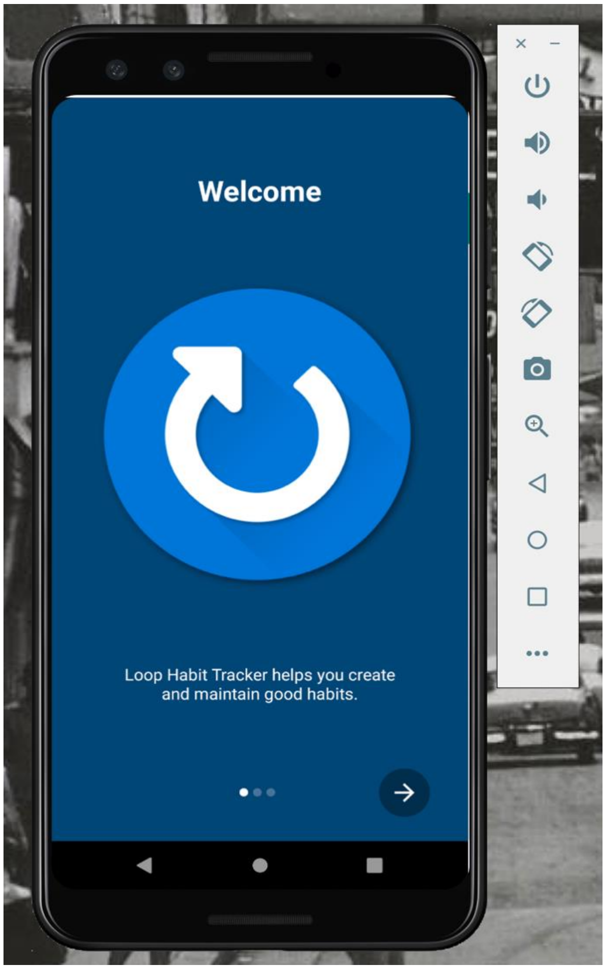
Traces Java and native code at the Android platform level.

Learn more ↗

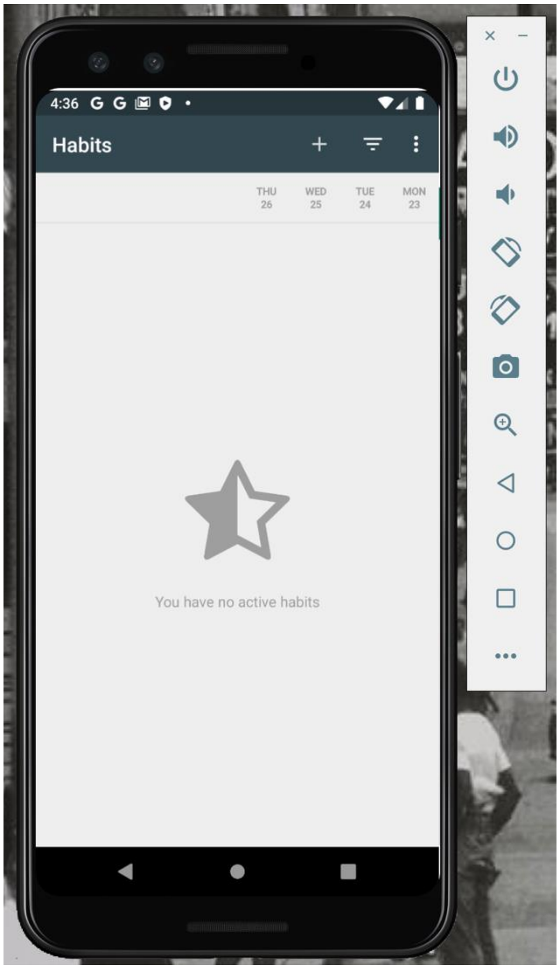
Success: Operation succeeded (3 minutes ago)

The performance tests were performed on Android Studio using the Android Profiler tool. A Google Pixel 4 phone emulator running Android API 26 was used to run these tests. We analyzed four instances of the app: launch, adding and editing an habit, rotating the phone while in the main activity, mark fulfillment of an habit, showing the dashboard related to a given habit and exporting the habits to a .csv file.

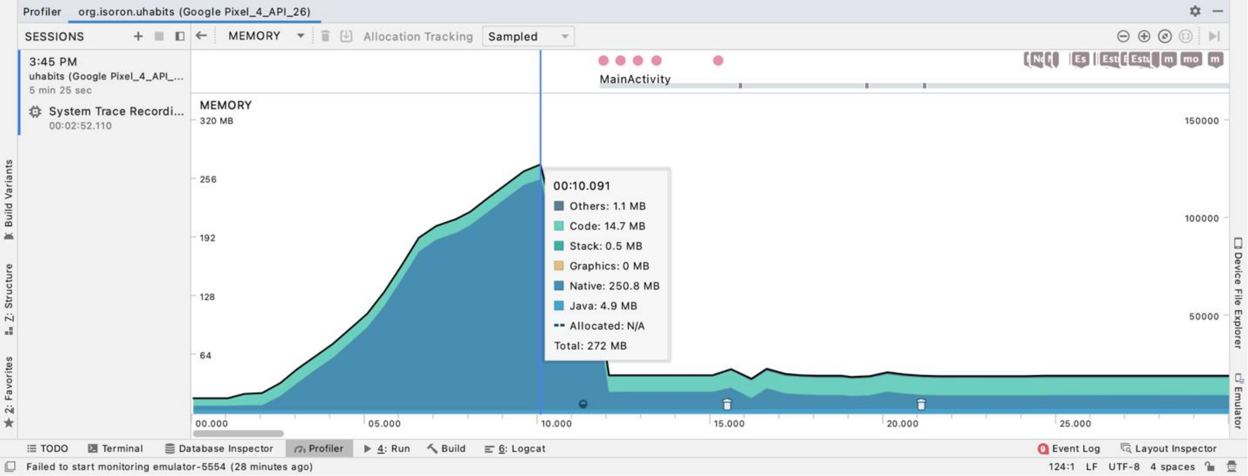
When launching the app, the first thing we observed is that it uses 25% of the CPU as we can see in the image below. After analyzing the trace recordings, we could find that there are three threads that are consuming most of the resources. These are: hwuiTask, .isoron.uhabits and RenderThread which are in charge of building the UI of the app and rendering it. However, given the low complexity of the interface of the app which we also show below, we consider that the amount of CPU used for building the UI is quite high and could lead to failures if for instance, when launching the app, the user is also running CPU consuming apps in the phone like Youtube, Skype or Whatsapp.


Welcome
Loop Habit Tracker helps you create and maintain good habits.

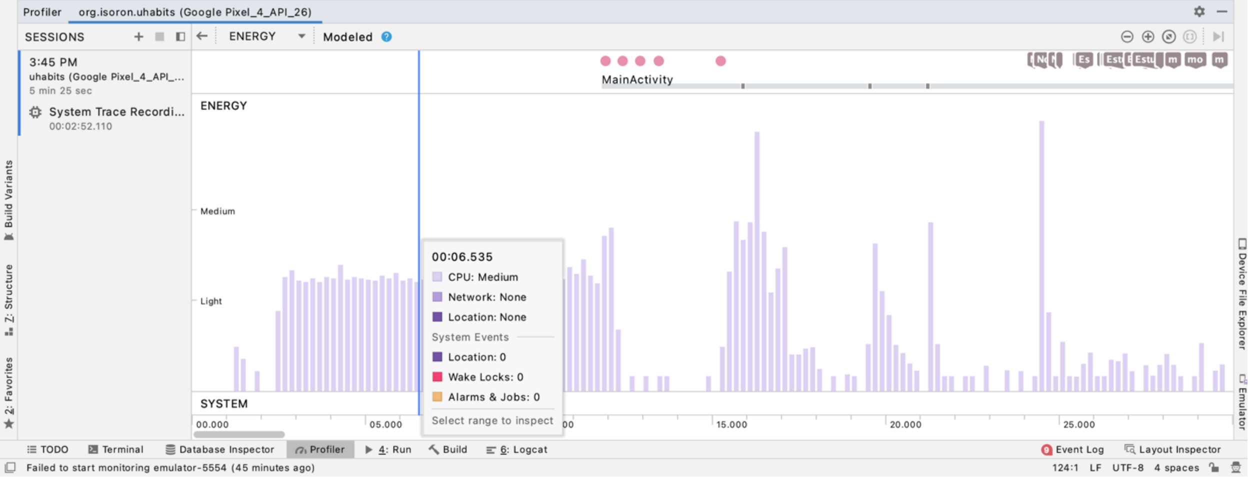
Ç



Following with the launch analysis of the app, we look at the memory usage. The maximum amount of memory used when launching the app was 272MB distributed as the image below show. One interesting thing to note is that graphics related task do not use any memory at all, but the UI rendering and launching are the most consuming CPU activities.

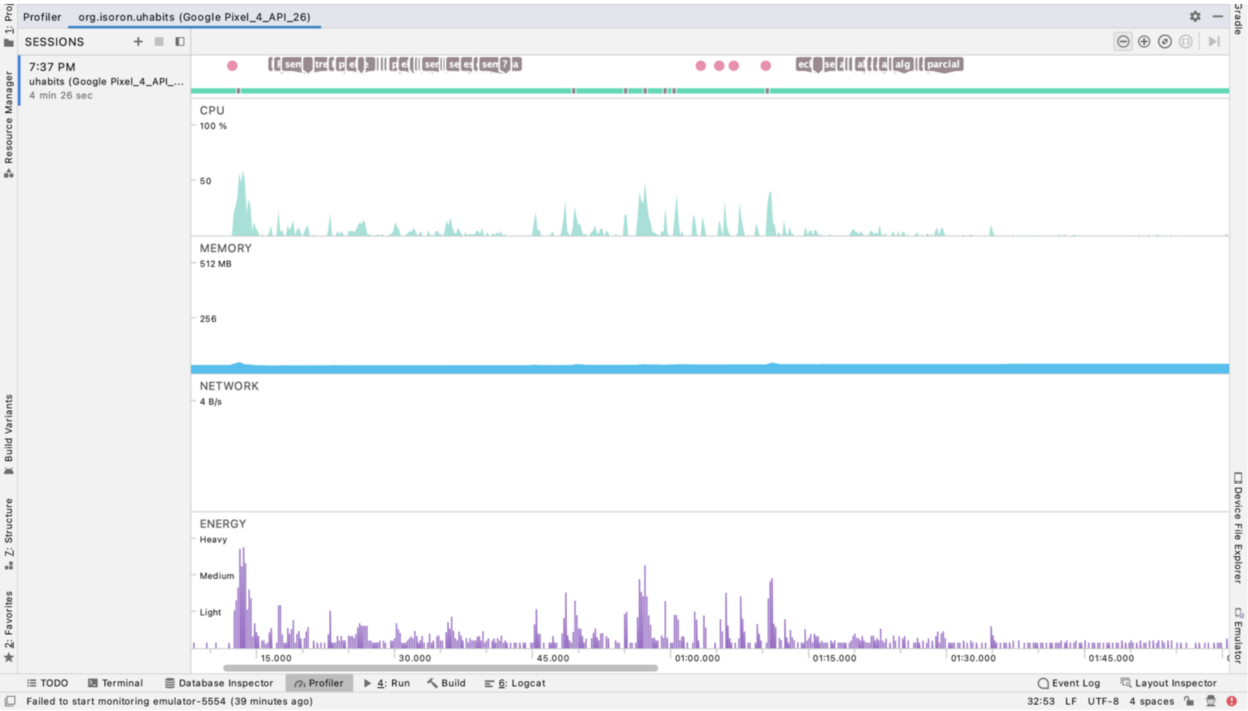


Given that the app does not provide any features that require internet connection, sent and received bytes during the launch and all the time we interact with the app where 0B/s. Moving on to the energy used during the launch of the app, Android profiler reported that energy used was on the MEDIUM range and that the CPU was the component that was using the largest amount of energy. This is consistent with what we discussed at the beginning of this section.



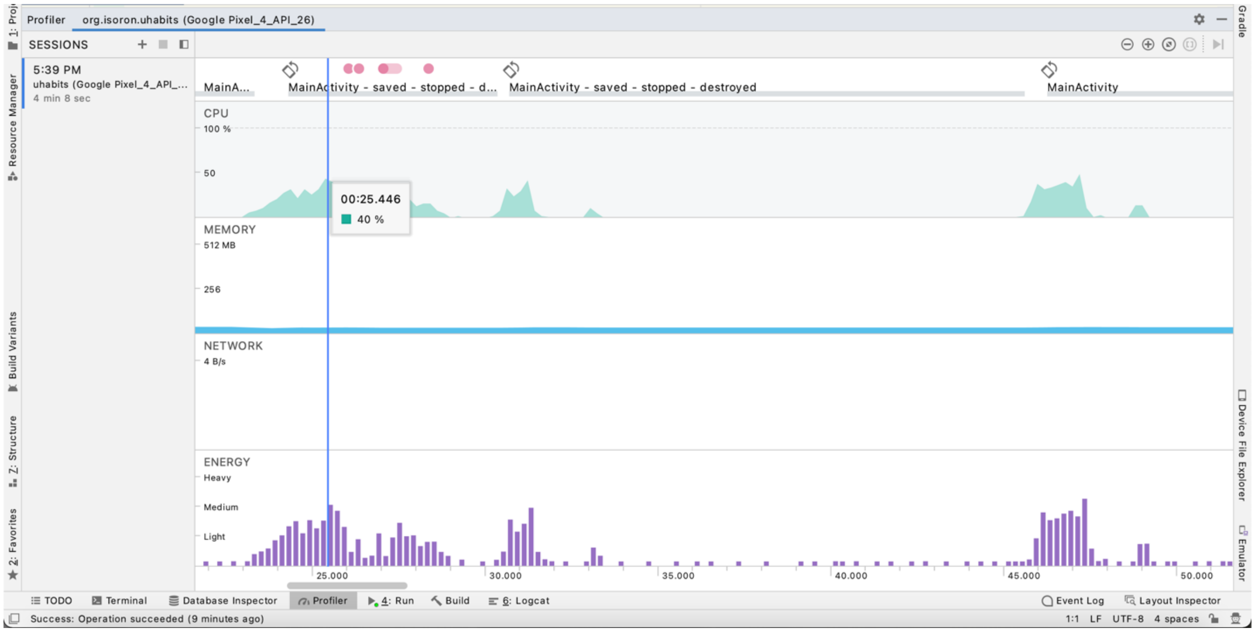
The next functionality that we evaluated during the performance test was the one of adding an habit. As with the launching, we kept track of the CPU, memory and energy usage when adding two habits with distinct characteristics. Regarding to the CPU usage it reaches its maximum of 60% when one touches the add habit button for the task. When editing the habit (setting its name, associated question, reminders), processor usage keeps below 20% most of the time. It again shoots up to 50% when one intends to change the color of the habit that one is adding.

When adding the second task, the app performs even better. For instance, when pressing again the button of add tasks, the CPU usage raises below 45%. The only thing worth of mentioning is that when setting the weekdays, in which the user wants to be reminded about its habit, which is done with a picker, the CPU usage raises to 50%. Memory allocation during the task adding keeps constant on the range of 42MB. Energy consumption, in accordance with what we have observed, was high when the add habit button was pressed and when selecting the color for the first habit that was added. On the second task that we added, the most energy consuming activity, as expected, was when the weekday picker displays. A screenshot of the profiler records is attached, as evidence, showing the performance of the app while adding the first habit.

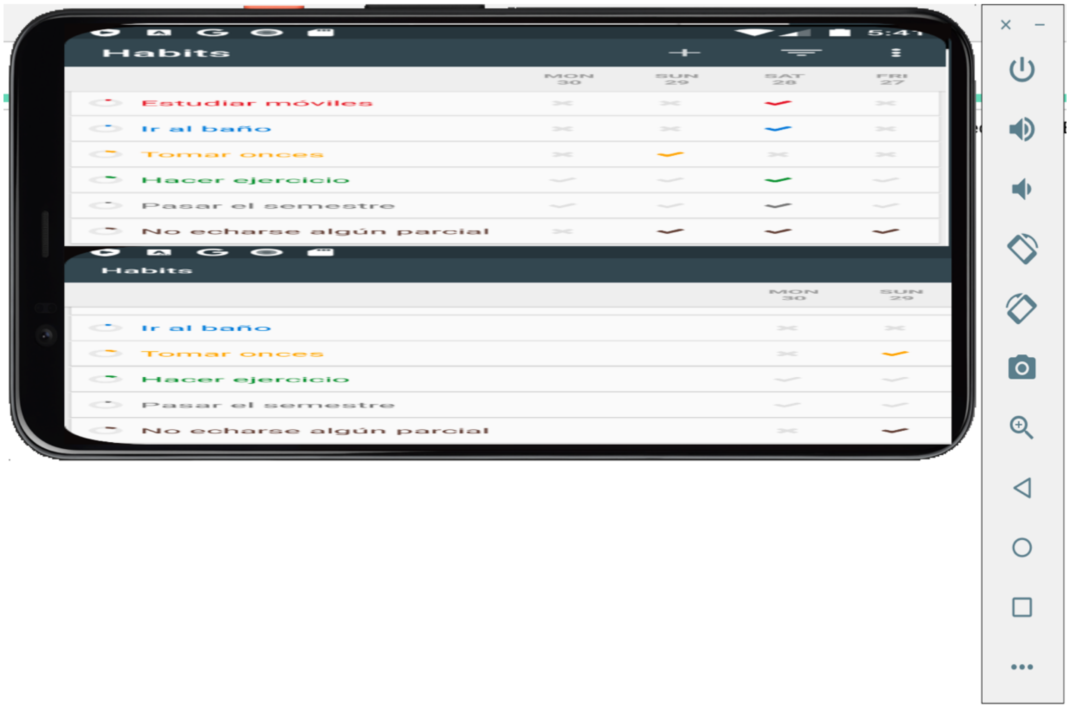


Failed to start monitoring emulator-5554 (39 minutes ago)

The third feature that we wanted to report is the one of rotating the phone on the main screen so that the user can have more days information of his habits. Regarding to the CPU and energy usage while rotating the phone, the levels raised to 49% and HEAVY energy consumption respectively which we consider high for an activity that just involves rotating the phone. The following screen capture is provided as evidence of what we are reporting.



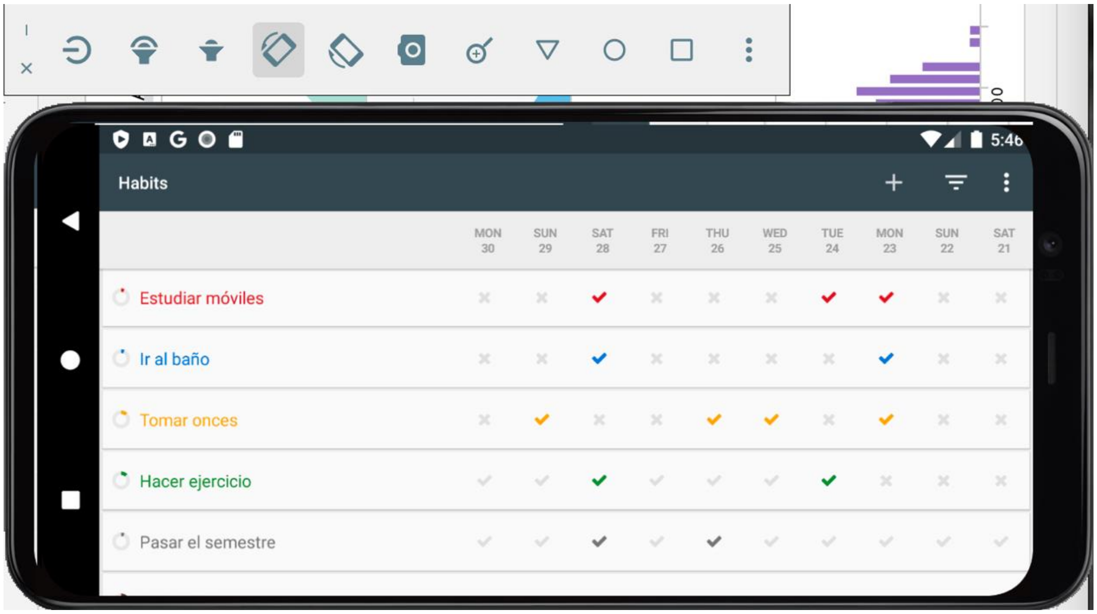
Moreover, when testing this feature, we discover a bug in the app. The view displayed when rotating the phone is not the one that the app is supposed to show. Sadly, we could not determine the origin of this bug in the app. Below we show the view displayed when rotating the phone and the view that the user is supposed to see.


Habits

	MON 30	SUN 29	SAT 28	FRI 27
Estudiar móviles	×	×	✓	×
Ir al baño	×	×	✓	×
Tomar onces	×	✓	×	×
Hacer ejercicio	✓	✓	✓	✓
Pasar el semestre	✓	✓	✓	✓
No echarse algún parcial	×	✓	✓	✓

Habits

	MON 30	SUN 29
Ir al baño	×	×
Tomar onces	×	✓
Hacer ejercicio	✓	✓
Pasar el semestre	✓	✓
No echarse algún parcial	×	✓

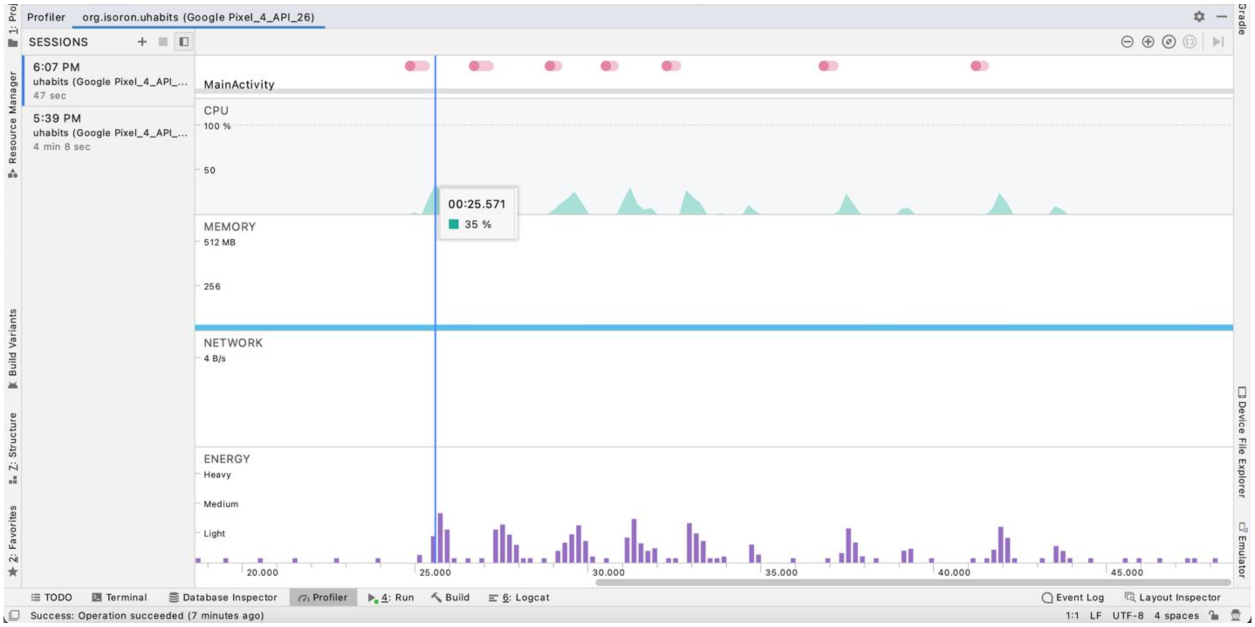


5:46

Habits

	MON 30	SUN 29	SAT 28	FRI 27	THU 26	WED 25	TUE 24	MON 23	SUN 22	SAT 21
Estudiar móviles	×	×	✓	×	×	×	✓	✓	×	×
Ir al baño	×	×	✓	×	×	×	×	✓	×	×
Tomar onces	×	✓	×	×	✓	✓	×	✓	×	×
Hacer ejercicio	✓	✓	✓	✓	✓	✓	✓	×	×	×
Pasar el semestre	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Following with the analysis of the features, we now discuss the one of marking the fulfillment of an habit in the main screen. The screen shot below shows the performance of the app when interacting with this feature of the application.



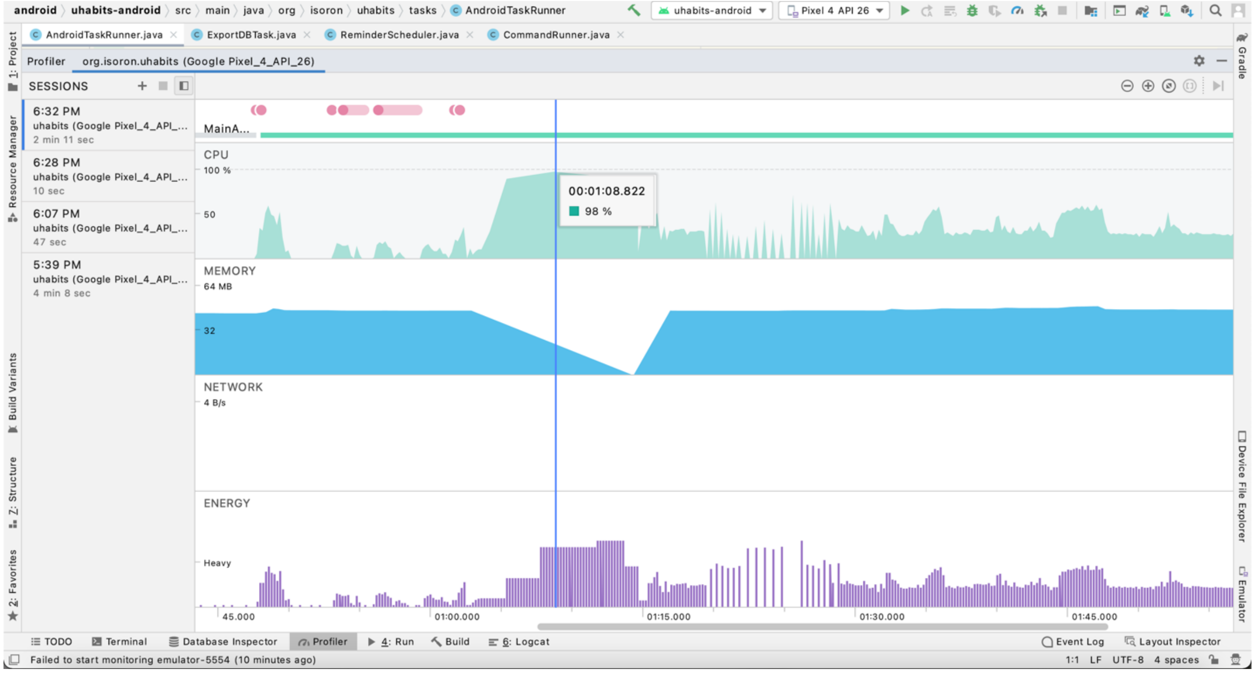
6:07 PM
uhabits (Google Pixel_4_API_...
47 sec

5:39 PM
uhabits (Google Pixel_4_API_...
4 min 8 sec

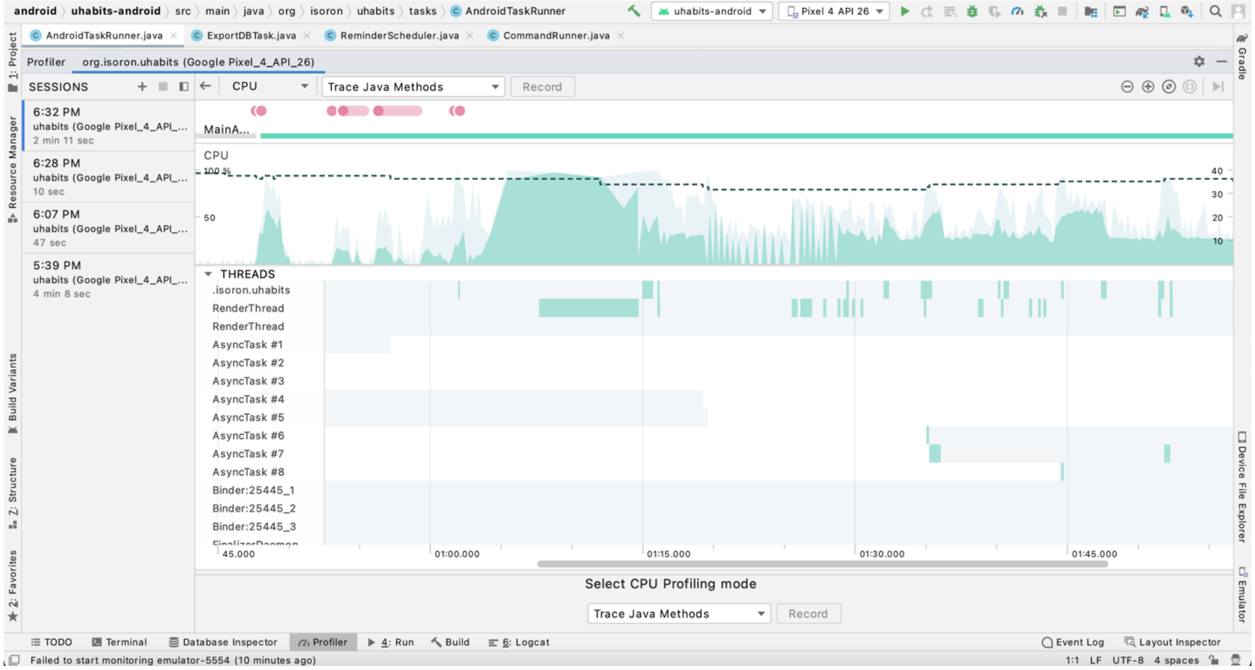
Success: Operation succeeded (7 minutes ago)

Every time we tapped the screen the CPU usage raises to, on average, 30% and the energy consumption levels where on the MEDIUM range. As with the rotation feature tested, we think these levels are high for a routine operation, like touching the screen is, in a mobile app.

The activity related to the display of the dashboard for a given habit is the worst performing of all. CPU usage and energy consumption levels raised to almost 90% and HIGH respectively while the dashboard was displayed.   We recognize that the rendering of the dashboard might be CPU expensive, but the levels that the app showed are beyond normal taking into account that the app does not offer interaction with the dashboard at all. At one point it was almost impossible to interact with the app since the screen just froze. After checking which was the thread that was consuming most of the resources, we find out that it was the RenderThread. Below are attached screen shots as evidence of what we have just reported.

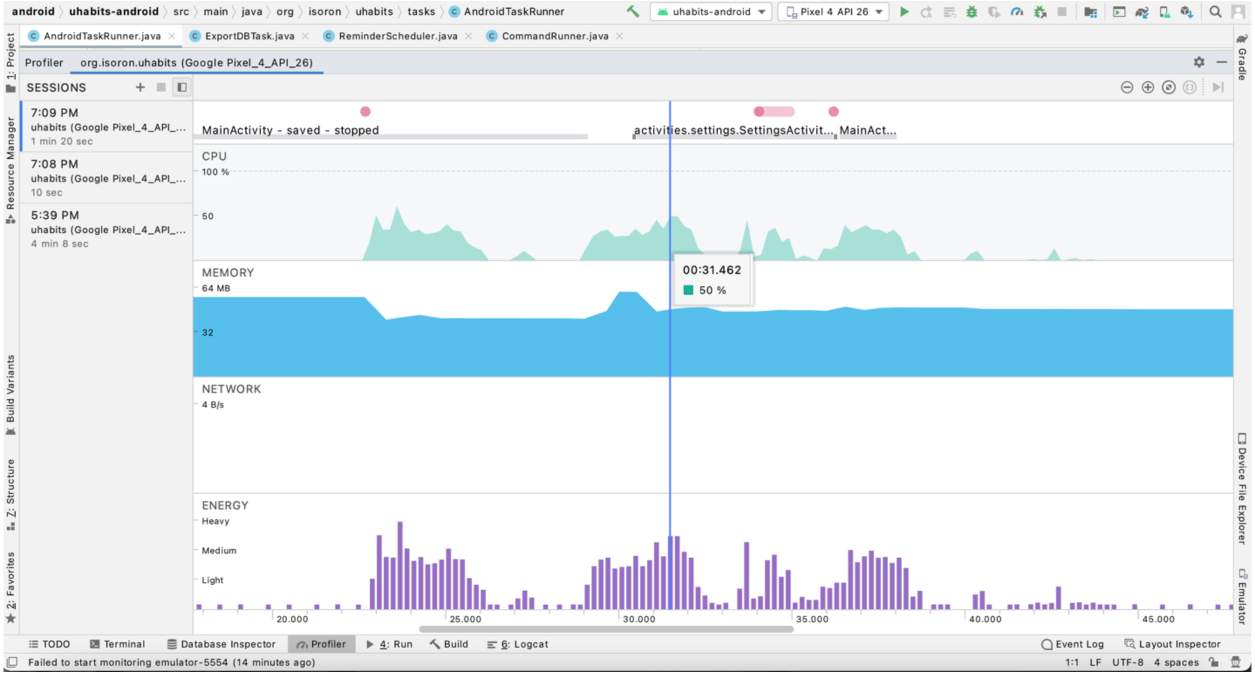


Failed to start monitoring emulator-5554 (10 minutes ago)



Failed to start monitoring emulator-5554 (10 minutes ago)

The last feature that we explored is the one that allows users to export to .csv their habits. This performs quite well. The app offers the possibility to send the .csv file generated via e-mail and we did not have any trouble during the process. It is worth mentioning, however, that when interacting with the settings view, the CPU and energy usage raised to 50% and HEAVY respectively. This maybe has to do with the rendering of the settings view of the app. Below, a screen shot is provided as evidence of what we just discussed.



Failed to start monitoring emulator-5554 (14 minutes ago)

As a conclusion for this section, we might say that, overall, the app does not perform well. It uses a lot of resources for routine processes in all mobile apps like loading a new view and responding to the interaction of the users with buttons and elements in screen. What is most concerning is that the app does not even offer any feature that has to do with network connection. The app UI, as we already reported, is not complex at all, however, the rendering of the views was one of the most resource consuming tasks. We do not recommend to use this app while using other ``heavy'' apps because it is highly probable that the system will crash due to the high use of the CPU.

3. Eventual connectivity

Regarding the functionalities of the application under situations where there is no internet or there is poor connectivity, the user will not realize any differences while using the app. This is because the app works using caching strategies that constantly use the local storage and shared preferences to store the user's data (their habits and notifications preferences mostly).

However, since the app has very limited functionalities that rely on internet (sharing your habits is the only one that needs internet) we are going to propose a little addition to the applications so that the habits can be also stored in a database located in the cloud.

3.1 Cloud data base proposal for Habit tracker

3.1.1 Create a Mongo DB Atlas cluster

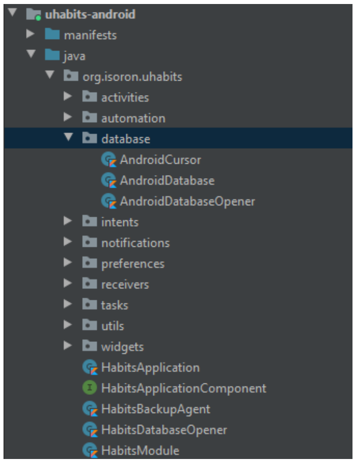
Here we are going to store only the data that is significant to the core of the applications (The habits) so there is no need of too many relationships. A NoSQL document database works perfectly here since we do not need complex transactions.

3.1.2 Create a Mongo DB Stitch app with the cluster

To create the connection between the app and mongo DB we need to create a stitch application using the previously created cluster.

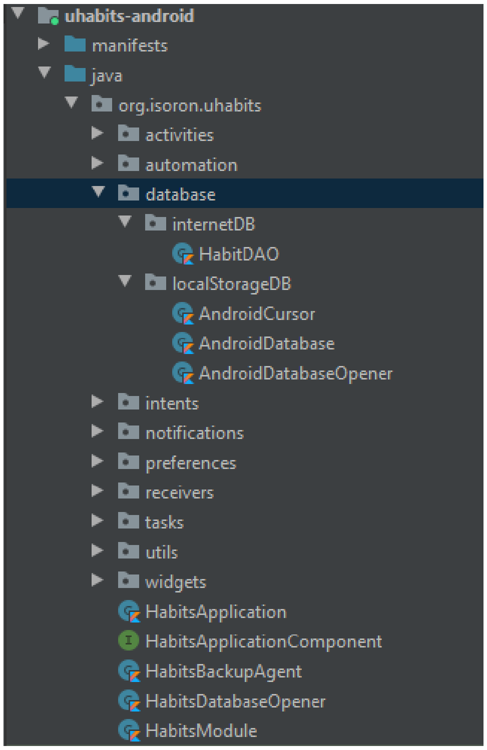
3.1.3 Make the classes needed to connect to the DB

In order to not change drastically the current folder structure and design patterns they have stablished; we would need to make additions to the database folder (which currently only supports SQLite). In this case the DAO pattern would work perfectly since we only need to create a class that handles the access to the database and performs queries to it.



```
▼ uhabits-android
  ▶ manifests
  ▼ java
    ▼ org.isoron.uhabits
      ▶ activities
      ▶ automation
      ▼ database
          AndroidCursor
          AndroidDatabase
          AndroidDatabaseOpener
      ▶ intents
      ▶ notifications
      ▶ preferences
      ▶ receivers
      ▶ tasks
      ▶ utils
      ▶ widgets
        HabitsApplication
        HabitsApplicationComponent
        HabitsBackupAgent
        HabitsDatabaseOpener
        HabitsModule
```

Figure 1 Old structure

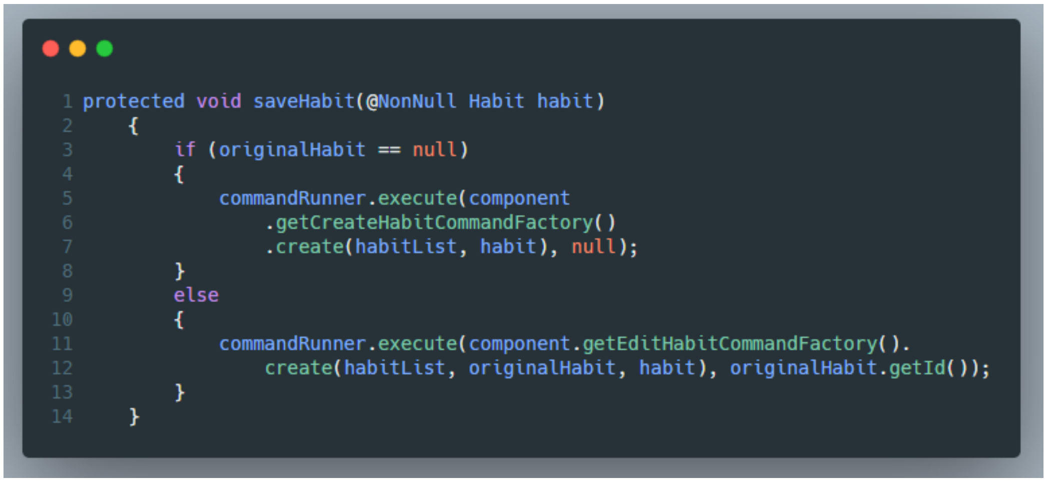


```
uhabits-android
    manifests
    java
        org.isoron.uhabits
            activities
            automation
            database
                internetDB
                    HabitDAO
                localStorageDB
                    AndroidCursor
                    AndroidDatabase
                    AndroidDatabaseOpener
            intents
            notifications
            preferences
            receivers
            tasks
            utils
            widgets
            HabitsApplication
            HabitsApplicationComponent
            HabitsBackupAgent
            HabitsDatabaseOpener
            HabitsModule
```

Figure 2 New Structure

3.1.4 Set up eventual connectivity logic

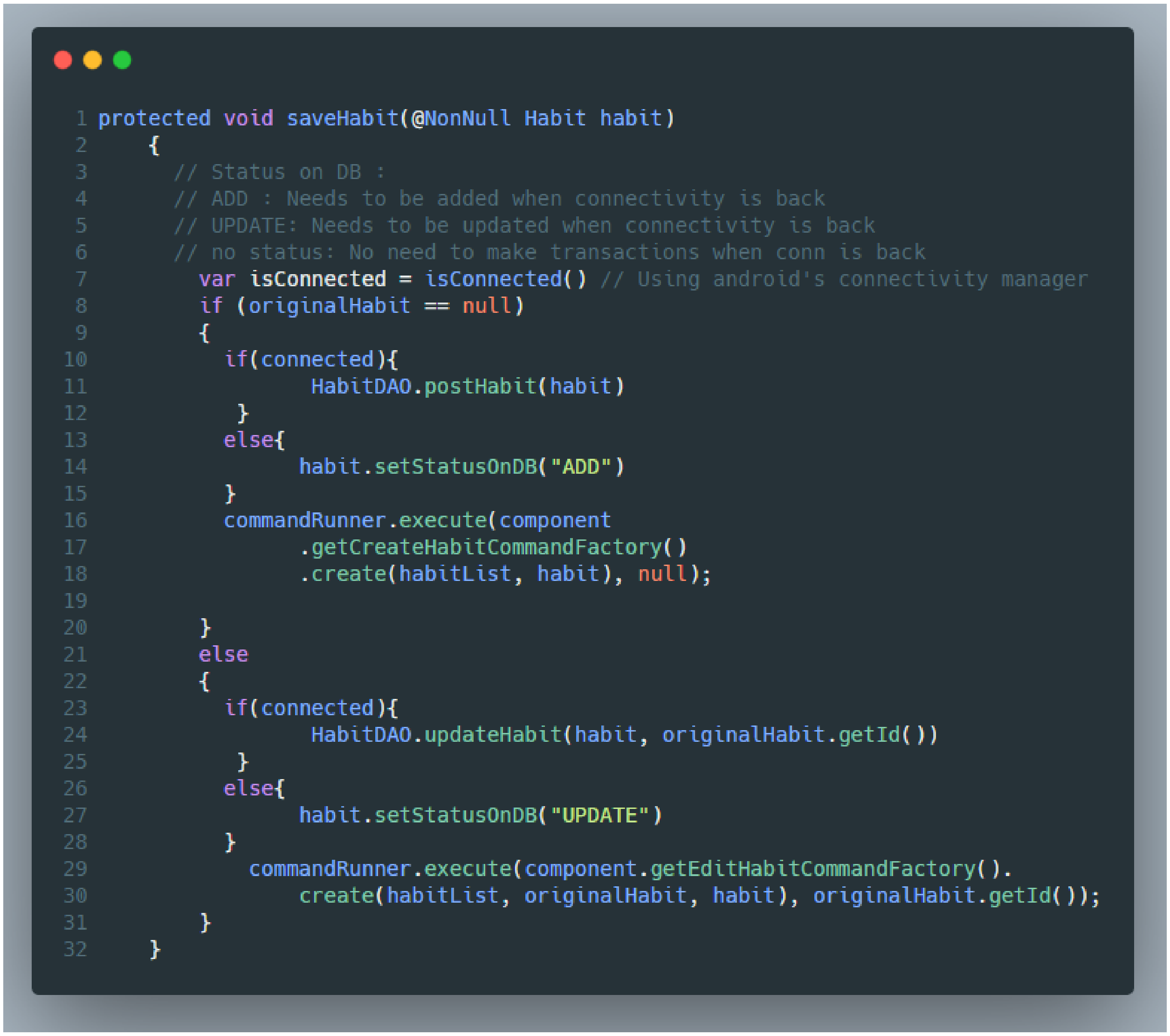
Now that we can connect to the database, we need to make the app aware of the internet status when creating, removing or editing a habit: For example, for editing a habit/ adding, we have the following code:



```
protected void saveHabit(@NonNull Habit habit)
    {
        if (originalHabit == null)
        {
            commandRunner.execute(component
                .getCreateHabitCommandFactory()
                .create(habitList, habit), null);
        }
        else
        {
            commandRunner.execute(component.getEditHabitCommandFactory().
                create(habitList, originalHabit, habit), originalHabit.getId());
        }
    }
```

Figure 3 Old add/edit habit function

Now, we would change it to:



```
protected void saveHabit(@NonNull Habit habit)
    {
      // Status on DB :
      // ADD : Needs to be added when connectivity is back
      // UPDATE: Needs to be updated when connectivity is back
      // no status: No need to make transactions when conn is back
        var isConnected = isConnected() // Using android's connectivity manager
        if (originalHabit == null)
        {
          if(connected){
                 HabitDAO.postHabit(habit)
           }
          else{
                habit.setStatusOnDB("ADD")
          }
          commandRunner.execute(component
                .getCreateHabitCommandFactory()
                .create(habitList, habit), null);

        }
        else
        {
          if(connected){
                 HabitDAO.updateHabit(habit, originalHabit.getId())
           }
          else{
                habit.setStatusOnDB("UPDATE")
          }
            commandRunner.execute(component.getEditHabitCommandFactory().
                create(habitList, originalHabit, habit), originalHabit.getId());
        }
    }
```

Figure 4 New habit add / edit function

Now the function has connectivity awareness and the correct persistence of the habits to a cloud DB using the previously created DAO.

Finally in order to make the application run those pending transactions we would use the current behavior of the application: Each time that the applications is opened, the list of habits gets iterated in order to be placed on the view, and now with the given the current status of each habit and the connection status, we would make the necessary transactions to the mongo DB cluster.

3.2 Overview Eventual connectivity

Even though currently Loop Habit tracker works perfectly with out or with internet, the app could improve easily integrating what they already have with a cloud data base, with out affecting their current functionalities and accessing to global data analysis of the user's habits and cloud support to their clients.

4. Caching

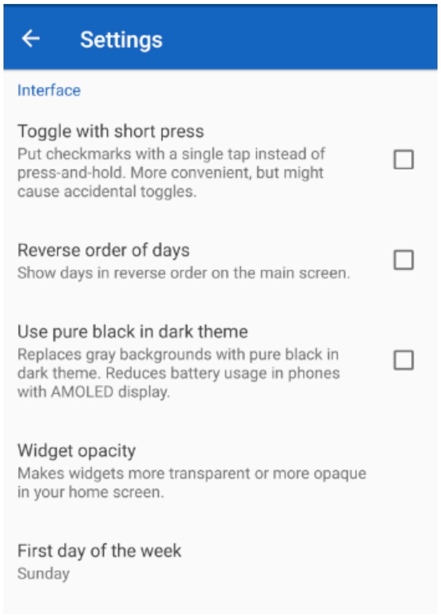
In order to understand the caching of Habit tracker it is necessary to go over the data that the have chosen to store locally on the phone:

4.1 SQLite habit storage

They store both habits and checks (days where the user completed a habit) using SQLlite. All of this is located on a package named database and it uses a class called HabitFactoy that uses the factory pattern in order to create or edit patterns (as java objects) that then are transformed into a object that SQLite understands.

4.2 Shared preferences

Loop habit tracker has quite a lot of different settings that the users can adjust. All of those are stored using Shared preferences on a package with the same name. Here you can check all the different settings they currently have.



Settings

Interface

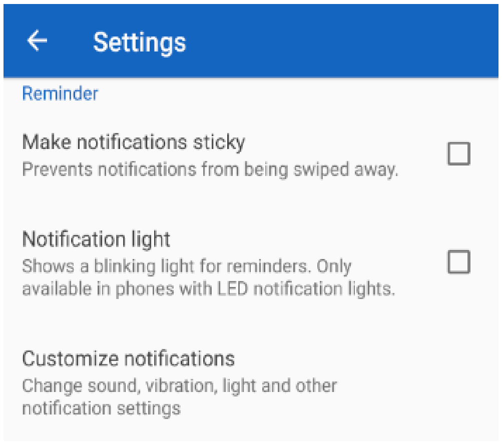
Toggle with short press
Put checkmarks with a single tap instead of press-and-hold. More convenient, but might cause accidental toggles. □

Reverse order of days
Show days in reverse order on the main screen. □

Use pure black in dark theme
Replaces gray backgrounds with pure black in dark theme. Reduces battery usage in phones with AMOLED display. □

Widget opacity
Makes widgets more transparent or more opaque in your home screen.

First day of the week
Sunday



Settings

Reminder

Make notifications sticky
Prevents notifications from being swiped away. □

Notification light
Shows a blinking light for reminders. Only available in phones with LED notification lights. □

Customize notifications
Change sound, vibration, light and other notification settings

For the caching of the applications we will go over the retrieval strategies, the fetching strategies and the caching data structures that are most used throughout the whole applications.

4.3 Caching retrieving

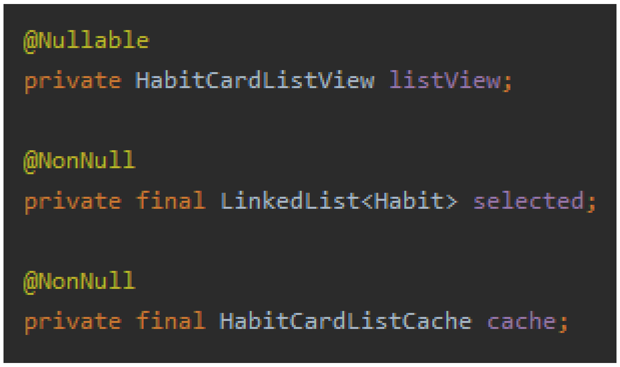
The application uses an always offline strategy for the most part, where access to the internet is mostly optional and sometimes not even called upon using the main functionalities. More specifically the application uses Cache, falling back to network.

4.4 Data fetching

The app never truly fetches the data from any backend service since it works mostly offline. Regarding the proposal we made on the eventual connectivity chapter, if we were to implement that, then we would suggest a “at launch time in background” strategy in order to update the cached data.

4.5 Caching data structures

The developers decided to use a linked list (using Kotlin) in order to manage the habits (which is the most important data entity).

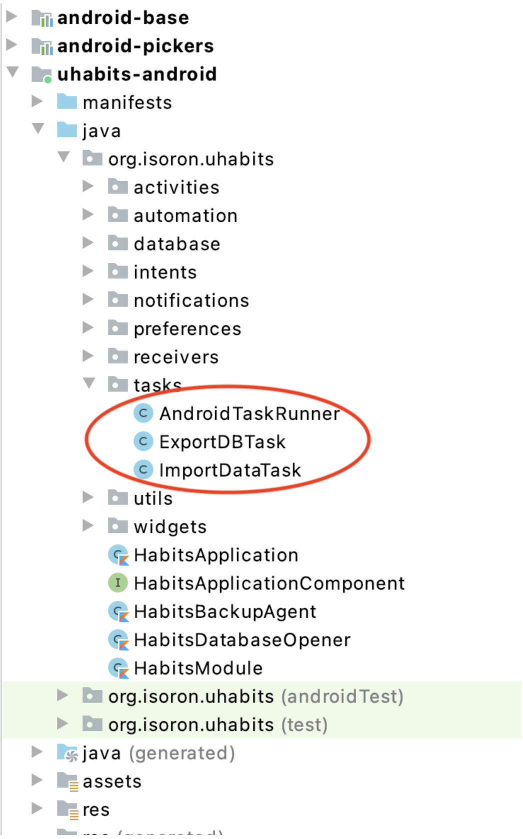


```
@Nullable
private HabitCardListView listView;

@NonNull
private final LinkedList<Habit> selected;

@NonNull
private final HabitCardListCache cache;
```

This is located on the Card list adapter that they created to show the list of habits the user creates.

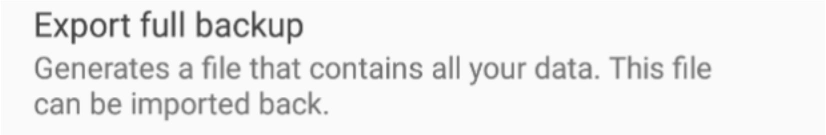
5. Multi - threading

All the multithreading of the app is handled in the task package of the app. More concretely in the classes shown below.



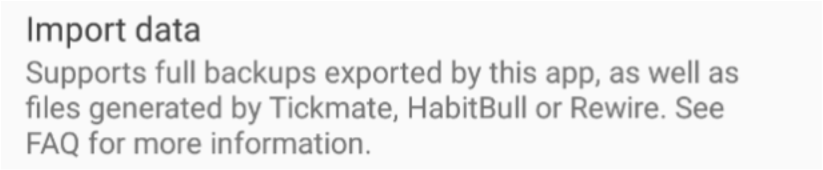
```
▶ android-base
▶ android-pickers
▼ uhabits-android
  ▶ manifests
  ▼ java
    ▼ org.isoron.uhabits
      ▶ activities
      ▶ automation
      ▶ database
      ▶ intents
      ▶ notifications
      ▶ preferences
      ▶ receivers
      ▼ tasks
          AndroidTaskRunner
          ExportDBTask
          ImportDataTask
      ▶ utils
      ▶ widgets
        HabitsApplication
        HabitsApplicationComponent
        HabitsBackupAgent
        HabitsDatabaseOpener
        HabitsModule
    ▶ org.isoron.uhabits (androidTest)
    ▶ org.isoron.uhabits (test)
  ▶ java (generated)
  ▶ assets
  ▶ res
```

Both ExportDBTask and ImportDataTask correspond to async tasks that do not run on the main thread. The first one deals with exporting a full backup of the habits that you have added in the application.



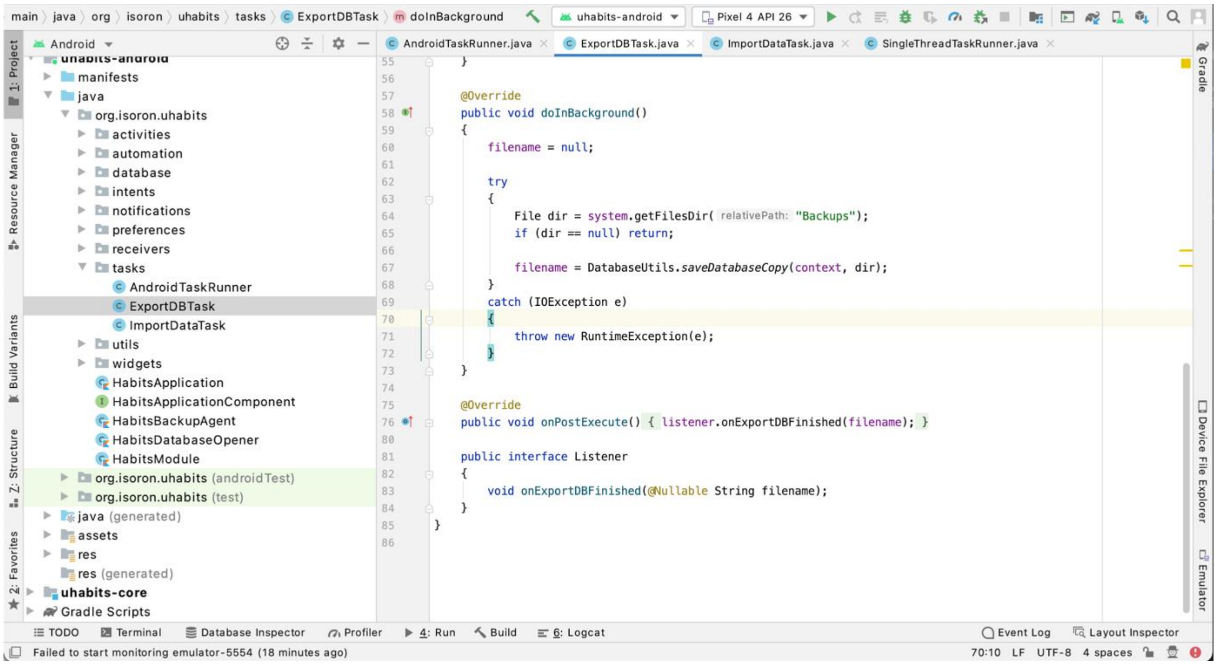
Export full backup
Generates a file that contains all your data. This file can be imported back.

The second class is in charge of importing into the app previously saved backups of the application.



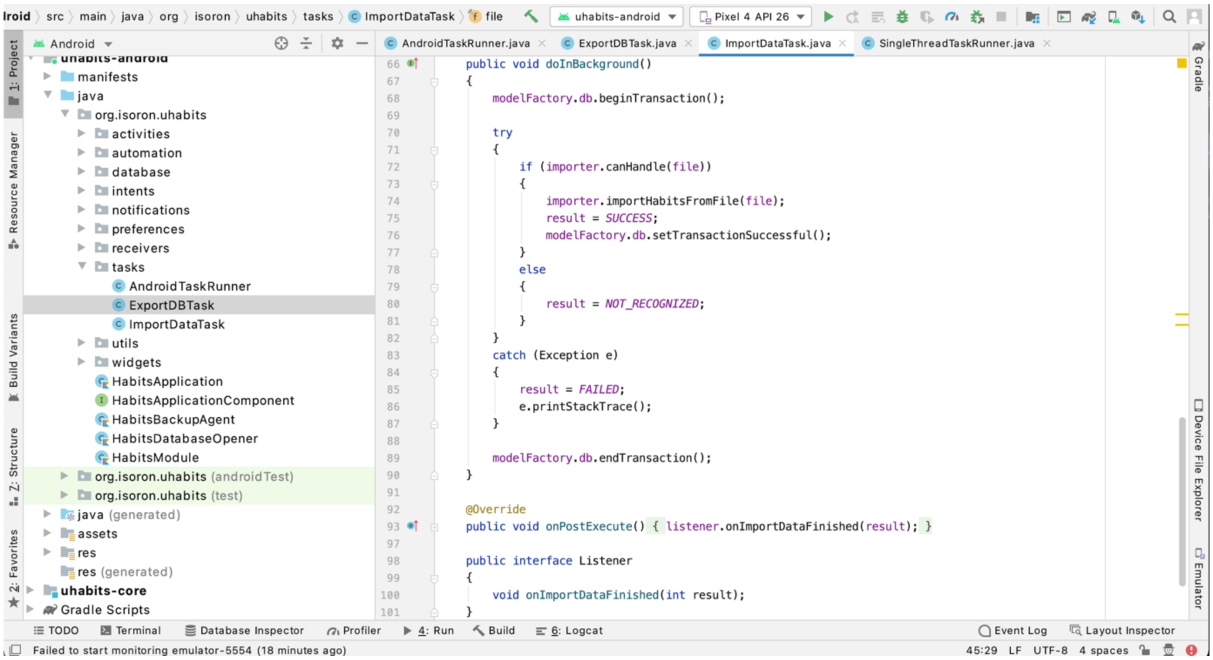
Import data
Supports full backups exported by this app, as well as files generated by Tickmate, HabitBull or Rewire. See FAQ for more information.

We find that these classes represent asynchronous tasks because they implement the doInBackground() and onPostExecute() methods which are typically implemented in tasks that do not run on the main thread. Below we provide screenshots of the implementation of these methods in ExportDBTask and ImportDataTask respectively.



```java
    }

    @Override
    public void doInBackground()
    {
        filename = null;

        try
        {
            File dir = system.getFilesDir( relativePath: "Backups");
            if (dir == null) return;

            filename = DatabaseUtils.saveDatabaseCopy(context, dir);
        }
        catch (IOException e)
        {
            throw new RuntimeException(e);
        }
    }

    @Override
    public void onPostExecute() { listener.onExportDBFinished(filename); }

    public interface Listener
    {
        void onExportDBFinished(@Nullable String filename);
    }
}
```

Failed to start monitoring emulator-5554 (18 minutes ago)



```java
    public void doInBackground()
    {
        modelFactory.db.beginTransaction();

        try
        {
            if (importer.canHandle(file))
            {
                importer.importHabitsFromFile(file);
                result = SUCCESS;
                modelFactory.db.setTransactionSuccessful();
            }
            else
            {
                result = NOT_RECOGNIZED;
            }
        }
        catch (Exception e)
        {
            result = FAILED;
            e.printStackTrace();
        }

        modelFactory.db.endTransaction();
    }

    @Override
    public void onPostExecute() { listener.onImportDataFinished(result); }

    public interface Listener
    {
        void onImportDataFinished(int result);
    }
```

Failed to start monitoring emulator-5554 (18 minutes ago)

Finally, we turn our attention to the AndroidTaskRunner class. This class is the implementation of the Façade pattern since it provides a unified high-level interface to both ExportDBTask and

ImportDataTask. We arrived to this conclusion when reviewing the implementation of this class and noting that it implements the CustomAsyncTask class

6. Memory management

7. Micro – optimizations

For this section we will go over the main micro-optimization's points learnt on the course:

7.1 Creating unnecessary objects on Code and on drawing Views.

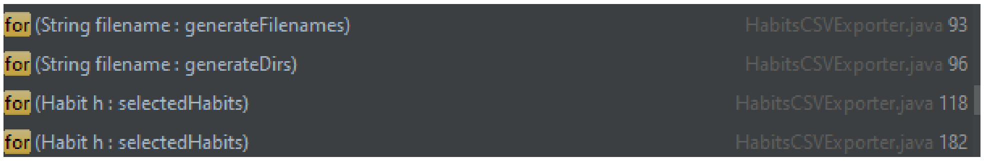
After going through the main classes of the code we found 0 Kotlin or Java warnings of variables no being used, and also after analyzing the loops on the code, there is no sign of creating auxiliary variables inside of the loop. There was also no warning on the linter analysis regarding this type of micro optimization.

7.2 Using primitive types over wrappers

In this case the app could see some improvement since it is overusing the class Integer instead using primitive int. We found over 34 classes that use Integer object for handling attributes that could be easily replaced with int. The same goes for Long which includes over 40 classes where the object Long is used instead of primitive long.

7.3 Indexed loops instead of iterators or for each's

Even though they use primarily indexed loops, they are overusing the for each method in some classes. In some other few cases they are using iterator of keys and values that could also be replaced with indexed loops in order to improve performance.



```
for (String filename : generateFilenames)        HabitsCSVExporter.java 93
for (String filename : generateDirs)             HabitsCSVExporter.java 96
for (Habit h : selectedHabits)                   HabitsCSVExporter.java 118
for (Habit h : selectedHabits)                   HabitsCSVExporter.java 182
```

7.4 Memory friendly data structures

As we explained on the caching section of the report, Habit tracker uses for the most part Linked Lists. Since this is a non-memory efficient data structure, we would recommend replacing all Linked lists with ArrayMaps. This data structure would improve the memory usage without compromising the response speed time that Linked lists provide currently to the app.

7.5 Avoid of the find by id method

This android method is only used 13 times (on 7 different activities) in the whole android Habit tracker project, for that reason it would not be recommended to make code changes to move to a binding pattern (it would not be worth it since it has very low usage).

7.6 Using relative or constraint layouts instead of nested linear layouts

We found 7 different layouts with nested linear layouts:

- date_picker_dialog.xml

- about.xml

- automation.xml

- edit_habit.xml

- edit_habit_name.xml

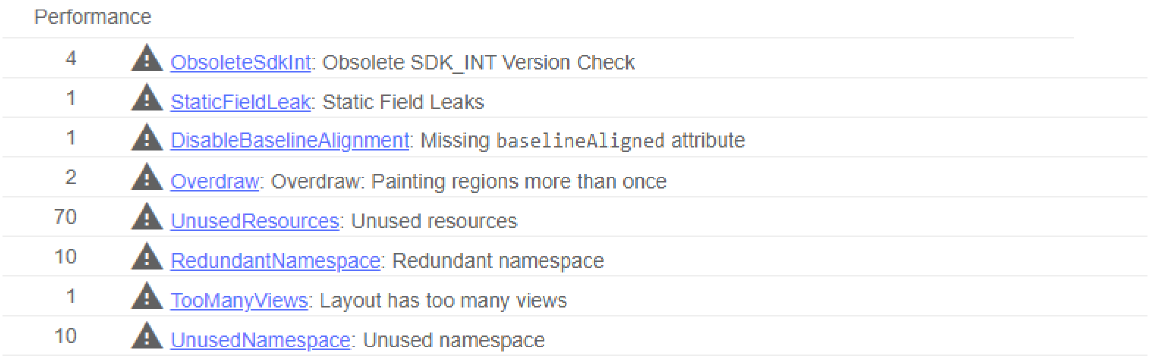
- edit_habit_reminder

- show_habit_overview.xml

In order to make the application more optimal Habit tracker would need to change this specific layout using constrained or relative layouts.

7.7 Linter analysis

After running the default linter analysis using android studio, we found 201 erros and 312 warnings. They can be checked at the web paged designed for the report. Here are the ones related to performance:



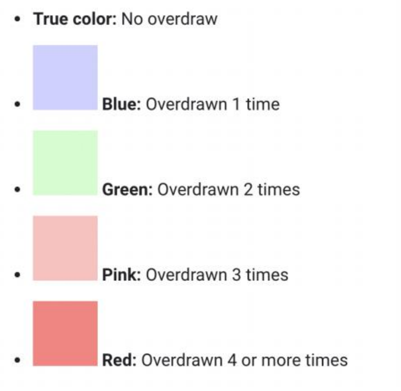
Performance		
4	⚠	ObsoleteSdkInt: Obsolete SDK_INT Version Check
1	⚠	StaticFieldLeak: Static Field Leaks
1	⚠	DisableBaselineAlignment: Missing `baselineAligned` attribute
2	⚠	Overdraw: Overdraw: Painting regions more than once
70	⚠	UnusedResources: Unused resources
10	⚠	RedundantNamespace: Redundant namespace
1	⚠	TooManyViews: Layout has too many views
10	⚠	UnusedNamespace: Unused namespace

The first 2 of them have a 6/10 priority score and the rest are scored with a 3 or a 1. Even though it seems there is not a significant warning, there are 99 warnings that together could translate into potential performance bugs.

7.8 Avoid overdrawing

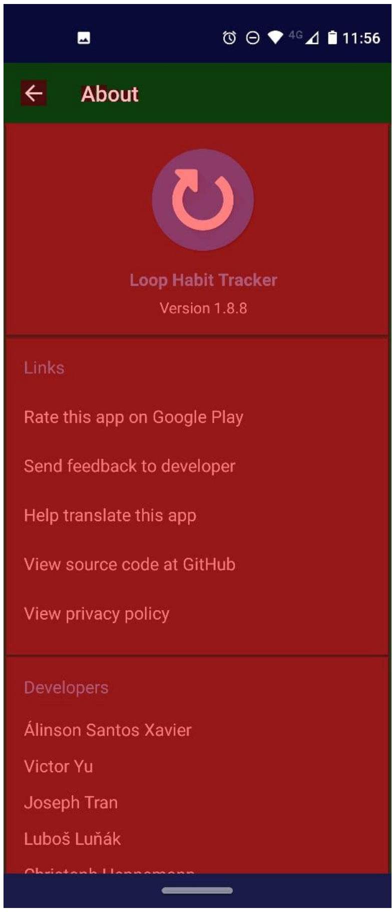
As shown earlier, there are two overdraw warnings. This can be found on the about.xml and the show_habt_inner.xml. We decided to run the app while using the android's GPU overdraw in order to check if the linter had missed some other overdrawn areas in the application:

To recap the color meaning of the Android's GPU overdraw utility:



- **True color:** No overdraw
- 🟦 **Blue:** Overdrawn 1 time
- 🟩 **Green:** Overdrawn 2 times
- ■ **Pink:** Overdrawn 3 times
- 🟥 **Red:** Overdrawn 4 or more times

Using the application with dark mode enabled:



11:56

About

Loop Habit Tracker

Version 1.8.8

Links

Rate this app on Google Play

Send feedback to developer

Help translate this app

View source code at GitHub

View privacy policy

Developers

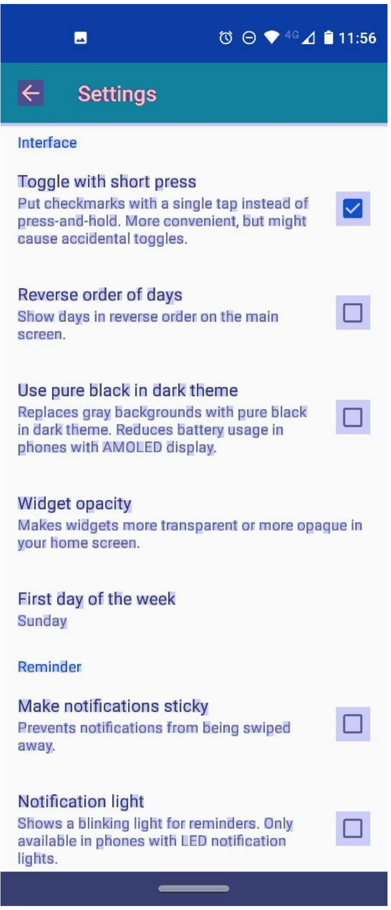
Álinson Santos Xavier

Victor Yu

Joseph Tran

Luboš Luňák

Christoph Hennemann



11:56

Settings

Interface

Toggle with short press ☑
Put checkmarks with a single tap instead of press-and-hold. More convenient, but might cause accidental toggles.

Reverse order of days ☐
Show days in reverse order on the main screen.

Use pure black in dark theme ☐
Replaces gray backgrounds with pure black in dark theme. Reduces battery usage in phones with AMOLED display.

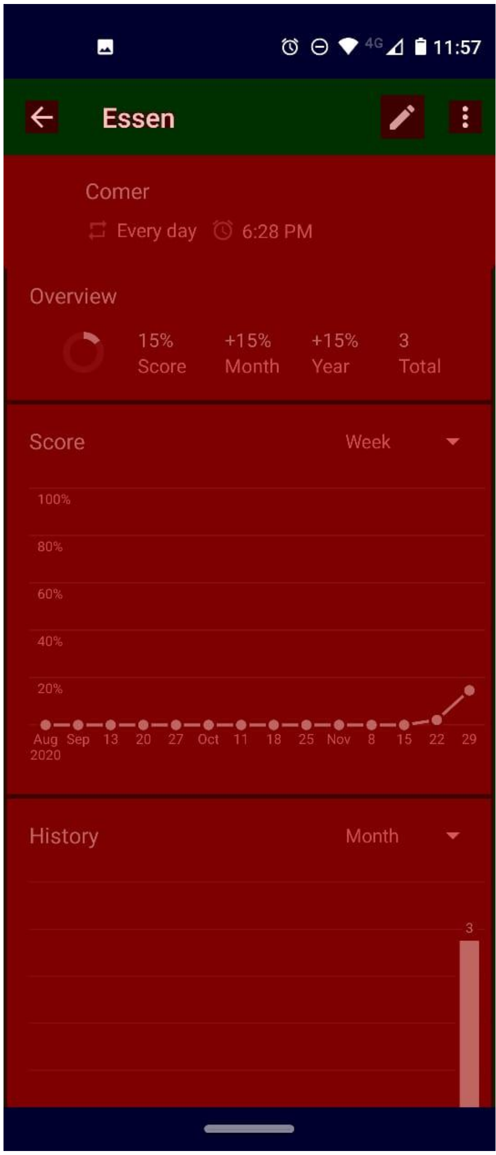
Widget opacity
Makes widgets more transparent or more opaque in your home screen.

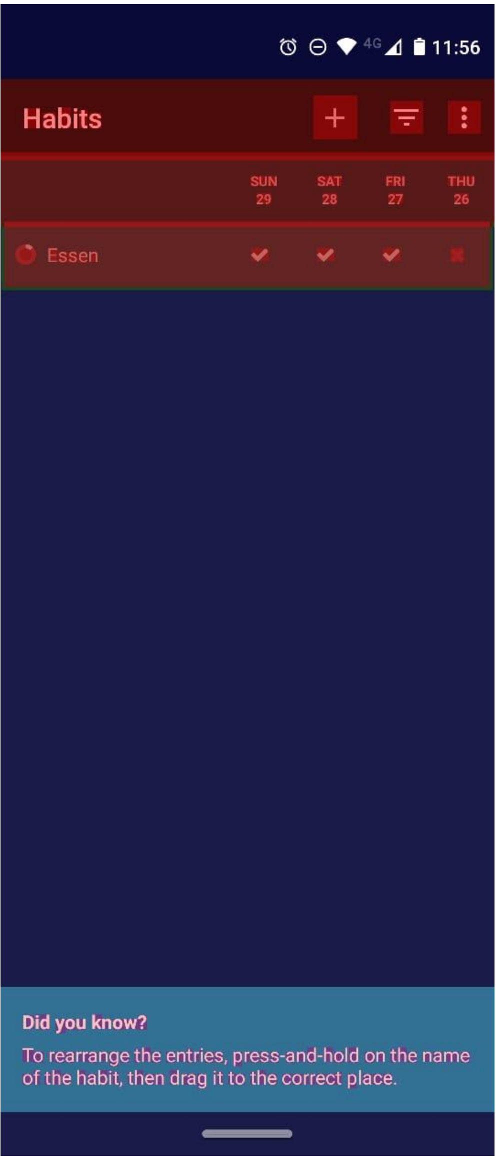
First day of the week
Sunday

Reminder

Make notifications sticky ☐
Prevents notifications from being swiped away.

Notification light ☐
Shows a blinking light for reminders. Only available in phones with LED notification lights.

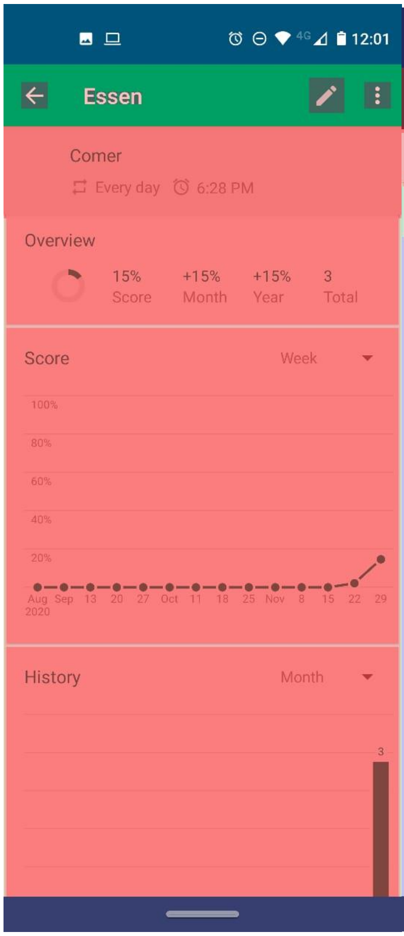


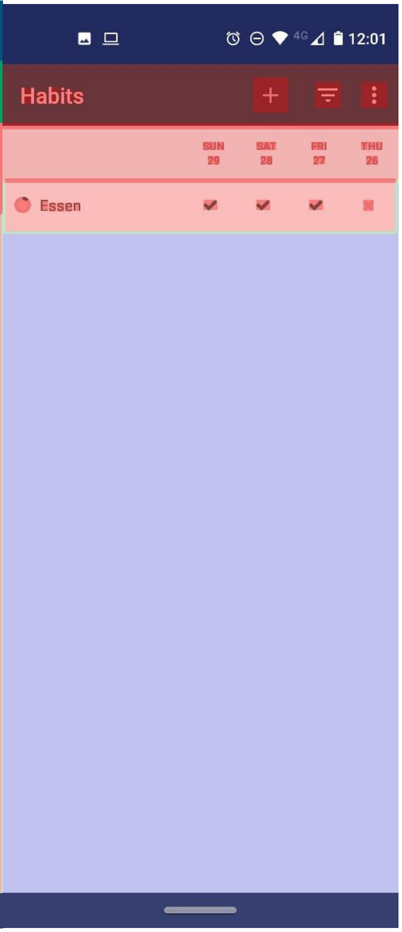


Did you know?

To rearrange the entries, press-and-hold on the name of the habit, then drag it to the correct place.

Using the application with dark mode disabled:





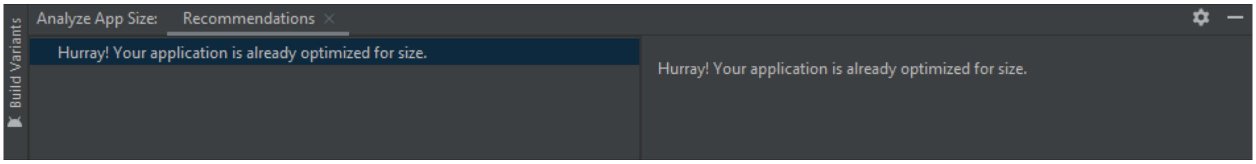
Running the application with the GPU overdraw tool activated clearly shows that the app is overdrawing constantly, and this could be causing performance issues or simply making the performance worse than it could be.

This is more significant using dark mode, where the overdraw happens up to 4 times or more in the detail habit activity and in the list of activities.

7.9 Reducing APK and bundle size

Habit tracker has a very small APK size, weighing roughly 3.4 MB. Also you can see that they have made an big effort in reducing bundle size as you can check that they have built their own pickers (time, color, calendar, etc.) instead of relying on third party packages.

Also after installing and running the google's app size analyser plug in, the results show the most optimal output, showing 0 recommendations in order to reduce the app's size:



Hurray! Your application is already optimized for size.

Hurray! Your application is already optimized for size.

7.10 Removing unused resources

As shown in the Linter analysis, there are 70 unused resources in the whole application. These are 70 small fix changes that could be easily done, and while doing so you would be improving the performance, as well as the memory usage of the application.

7.11 Overview micro-optimizations

Overall habit tracker has many places where it could do more in order to optimize its product, especially fixing overdrawing, removing some unused resources, using primitive types over objects and replacing nested linear layouts with constrained or relative ones. But in the other hand, the app makes a great work in its APK and bundle size, usage of weak references in multithreading, use of static in nested classes, good practices in lifecycle methods and very limited usage of the find view by id method.

8. Security

In this section we will go over the main security aspects to take into consideration in an android java / kotlin application:

8.1 Implicit intents

Habit tracker has one class than handles all the intents of the application (IntentFactory.kt). Even though most of them are explicit within the same application, however they have 6 different implicit intents that mostly redirect to webpages with different information aspects of the app (FAQ, github repository, email, playstore, etc). These do not represent a direct security tread, however the user might without knowing decide to open the web page or email with an untrusted application, and therefore creating a possible security issue.

8.2 Data issues

Each habit and its related data are stored in a SQLite database on the phone's local storage. On a first analysis this would mean that this data is safe, however since Android still has a long way to go in order to make local storage fully secured, we know it's xossible for another malicious app to force its way to the SQLite database and get those habits data from the user. This could be fixed using SQLCipher or using the SQLite version that secures the data (however this would need payment which probably would harm the current business model that Loop habit tracker has).

Regarding the logs, the app has 60 of them throughout the whole project, however its purpose is mostly for performance profiling, debugging, checking the status of systems, etc. So there is not any sensitive data getting leaked through there.

Furthermore, even though the shared preferences are not encrypted using the EncryptedSharedPreferences class, we do not consider this to be a security issue, since the data stored using this type of local storage are related to very basic settings that do not represent anything sensible or private.

8.3 Exported components

In the app's manifest can be found 9 activities, 8 receivers and 1 provider. Only 2 activities and 1 receiver have the android exported attribute set as true, mainly to manage notifications, email, and widget related functionalities. Following good practices, the provider has explicitly set the same attribute in false, however the rest of activities and receivers could improve security practices by setting the attribute as false as well.

One of the activities that can be exported, is the one that handles the list of habits, so the data issues that were spoken about in the previous point, get much more problematic and impose a significant issue to the user's data (Even though there is no personal information such as name, age, etc; Your habits could be accessed from a third party that could sell or use that data without the user finding out).

References

- https://sourcemaking.com/design_patterns/facade
-